conference

proceedings

First Workshop on Industrial Experiences with Systems Software (WIESS 2000)

San Diego, California
October 22, 2000

Sponsored by
USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX
First Workshop on Industrial Experiences with Systems Software
San Diego, California
October 2000

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

The price is $18 for members and $24 for nonmembers.
Outside the U.S.A. and Canada, please add
$10 per copy for postage (via air printed matter).

ISBN 1-880446-15-4
Printed in the United States of America on 50% recycled paper, 10–15% post-consumer waste.

USENIX Association

Proceedings of the First Workshop on Industrial Experiences with Systems Software (WIESS 2000)

October 22, 2000
San Diego, California, USA

Conference Organizers

Program Chair

Dejan S. Milojicic, *HP Labs*

Program Committee

Gaurav Banga, *Network Appliance*
Mark R. Brown, *Microsoft*
Mark S. Brown, *IBM*
Eduard Bugnion, *VMware*
Rob Gingel, *Sun Microsystems*
Fred Glover, *Compaq*
Ira Greenberg, *Oracle*
Larry Huston, *Intel*
Rodger Lea, *Sony*
Udi Manber, *Yahoo*
Franklin Reynolds, *Nokia*
Toshi Sakuraba, *Hitachi*
Indira Subramanian, *HP*
Franco Travostino, *Nortel Networks*
Richard Wheeler, *EMC*

Steering Committee

Jean Bacon, *Cambridge University, SIGOPS Member*
Andrew Hume, *AT&T Research, USENIX Vice President*
Valerie Issarny, *INRIA, ACM SIGOPS Vice Chair*
Michael B. Jones, *Microsoft Research, IEEE TCOS Vice Chair*
Marshall Kirk McKusick, *Author and Consultant, member of USENIX Board of Directors*
Dejan S. Milojicic, *HP Labs, IEEE-CS TCOS Chair*

Publications Chair

Alan Messer, *HP Labs*

The USENIX Association Staff

First Workshop on Industrial Experiences with Systems Software

October 22, 2000
San Diego, California, USA

Message from the Workshop Chair

The WIESS workshop was conceived as a gathering of papers and people from the industry at an event featuring work "from the trenches." It was designed to complement the SOSP and OSDI symposia by focusing on industry results of immediate benefit and use rather than long-term research. The workshop was meant to be informal in style, featuring short papers, invited talks, and posters which would allow the participants to focus on practical aspects and current problems in the field. In retrospect, we believe that we have achieved our planned goal. WIESS features papers and posters describing a system from the airline industry, computer and storage systems, and various tools, including those for ia64 architecture. Distinguished invited speakers, such as James Gosling and Steven Kleiman, and a panel discussion on the research and technology transfer with Rob Pike, Andrew Tanenbaum, Kirk McKusick, and Rob Gingell, round out the program.

There were 18 papers officially submitted, all of which underwent review by at least 5 program committee members. The WIESS PC members all come from industry. I was the single member from the Labs. The entire PC had access to all papers and all reviews for each paper. In the case of PC members who also submitted papers, the identity of the reviewers of their papers was not disclosed. PC members did not review papers from their companies. We met at the PC meeting and went over all papers, sorting them into three categories: accepted, posters, and rejected papers. We accepted 8 regular papers and 5 posters. (Unfortunately, one poster presenter had to withdraw.) Each accepted paper or poster was assigned a shepherd. Only after approval by the shepherd were these papers finally accepted for publication. The poster session papers were allowed two pages in the proceedings and offered an opportunity to present their work as a poster at the meeting. The reviewing process was conducted using the Wimpe tool written by David Nicol of Dartmouth College. This tool has been used by many events in the past. I thank David Nicol for including many changes that John Wilkes and myself provided as a feedback from using the tool as chairs of SOSP '99 and ASA/MA '99, respectively.

Most of the submitted papers were in the operating systems category (11), programming environments (8), tools (7), and high availability (6). The other represented areas were real-time (4), distributed systems (3) embedded systems (3), middleware (3), fault tolerance (2), networking (2), security (2), and system administration (2). The accepted papers are classified into three categories: system architecture, performance, and tools. Overall, we believe that we achieved our original goal to select papers describing real systems and experience from the trenches. We are particularly happy about the tools-related papers, for such papers have generally been underrepresented at most system-related events. We hope this new trend will continue.

I would like to express my thanks to the steering committee, the program committee, and the external reviewers. We expended a lot of effort in paving the road for future events of this kind by setting a standard of reviewing industrial papers and providing quality papers to the community. I believe that we have succeeded in this task, although the ultimate judgment is left to readers. This job was not easy, but I hope that we all enjoyed the process and learned a lot about our own field. I certainly have. I would also like to express my thanks to the organizer and sponsors of this events: USENIX, ACM SIGOPS, and IEEE TCOS. As usual, USENIX did a superb job of organizing the whole event. I keep being amazed at how the team led by Ellie Young manages to keep all USENIX-sponsored events on track. Thanks are due to Gale Berkowitz, Bleu Castañeda, Judy DesHarnais, Vanessa Fonseca, Barbara Freel, Jane-Ellen Long, Monica Ortiz, and Jennifer Radtke. Special thanks are due to EMC for sponsoring the event with their generous contribution, and to Ric Wheeler for making this happen. Finally, I would like to thank all the authors who submitted their papers. A few promising papers had to be rejected, but this event would not have been as good without their submissions.

Dejan S. Milojicic

WIESS Program Chair

Operational Information Systems - An Example from the Airline Industry

Van Oleson
Delta Technology
Georgia Tech
van.oleson@delta-air.com

Greg Eisenhaur
Georgia Institute of
Technology
eisen@cc.gatech.edu

Calton Pu
Georgia Institute of
Technology
calton@cc.gatech.edu

Karsten Schwan
Georgia Institute of
Technology
schwan@cc.gatech.edu

Beth Plale
Georgia Institute of
Technology
beth@cc.gatech.edu

Dick Amin
Delta Technology
dick.amin@delta-air.com

Abstract

Our research is motivated by the scaleability, availability, and extensibility challenges in deploying open systems based, enterprise operational applications. We present Delta's mid-tier Operational Information Systems (OIS) as an approach for leveraging its legacy operational OLTP infrastructure, to participate in the emerging world of electronic commerce, as well as enable new applications. The approach is to place minimally intrusive 'taps' into the legacy OLTP systems to capture transactions as they occur for consistent replay in the mid-tier OIS. One important issue addressed by our work is the processing, and dissemination of information in the mid-tier system itself, potentially serving hundreds of thousands of access and display points, distributed across a highly geographically distributed system (e.g. airports world wide), and also involving large `working sets' of operational data, used by applications that require rapid response and also rapid recovery from failures. To address the scaleability, availability, and cost of this OIS infrastructure, we are researching cluster computing techniques, as well as, devising replication and failover techniques. To address the communications scaleability requirements, we are experimenting with novel event-based implementations of information transport and processing, that include reliable multicast variations.

1. Introduction

Increased competition in the airline industry is stimulating the development of new applications of information technology, including a new strategic focus on electronic commerce at Delta Air Lines. Traditionally, large enterprise computing at companies like Delta has relied on using clusters of mainframes running proprietary information systems software. For example, Delta relies on a cluster of IBM S/390 mainframe computers running system TPF (Transaction Processing Facility), a specialized operating. These traditional online transaction processing systems (OLTP) support applications that automate the majority of the airline's operational services. The TPF and MVS systems architecture has proven to be highly scaleable and available, and the systems have operated successfully over the last 30 years and through the Y2K bug scare.

It is difficult to modify these existing OLTP applications to accommodate a changing business. Many of the applications were developed in assembly language and have evolved over a period of more than 30 years. Originally, the applications were designed to implement specific business models and offer little flexibility to support new business models and processes. Specifically, these applications maintain ownership of rigidly defined data sets, and their legacy data formats offer little opportunity for creating new relationships to other application data. Additionally, new business models result in new applications, some of which leverage the Internet. This exposes the legacy systems to unforeseen transaction volumes.

In response to these limitations, a novel strategy pursued by Delta is the addition of mid-tier enterprise

information systems, termed Operational Information Systems (OIS). The wealth of information in the existing OLTP systems is harvested by "grabbing" strategic transactions as they occur in soft real-time. These transactions are then replicated and consistently replayed in the newly introduced OIS. In this new environment, data resulting from the transactions is mapped into alternative evolvable formats, which is correlated with previously unrelated information, as well as information from sources other than the OLTP systems. Additionally, the immediate correlation stimulates events, which are derived from the transaction histories. This capability enables an entirely new class of real-time event based applications, which have proven to radically improve the efficiency of airline operations.

The new mid-tier OIS, considered in concert with the legacy OLTP system, is the basis on which Delta constructs new applications and improves current business operations, including improving the "Customer Experience". The key element to their success is the development of new mission-critical software and hardware infrastructures that support these efforts.

In the remainder of this paper, we first characterize Delta's OIS strategy and its components in more detail. We then state the issues that motivate the academic research in to highly scaleable and highly available OIS implementations.

2. OIS Components

The systems model in figure 1 depicts the overall architecture including the major systems and physical components that implement the OIS.

Legacy OLTP Systems are long-lived information systems that continue to support application operations. These applications are `tapped', which result in transaction histories to be distributed to the Event Derivation Engine.

Event Derivation Engine is a Global Information Base comprised of a set of servants that internalize transaction histories from OLTP systems, as well as, other internal and external sources of information. The EDE correlates and consolidates this information and maintains an operationally narrow subset, or operational window, from which it derives events for publication. Additionally, the maintained consolidated information serves as a base for simple request and replies, as well as, initial states for subscribing clients.

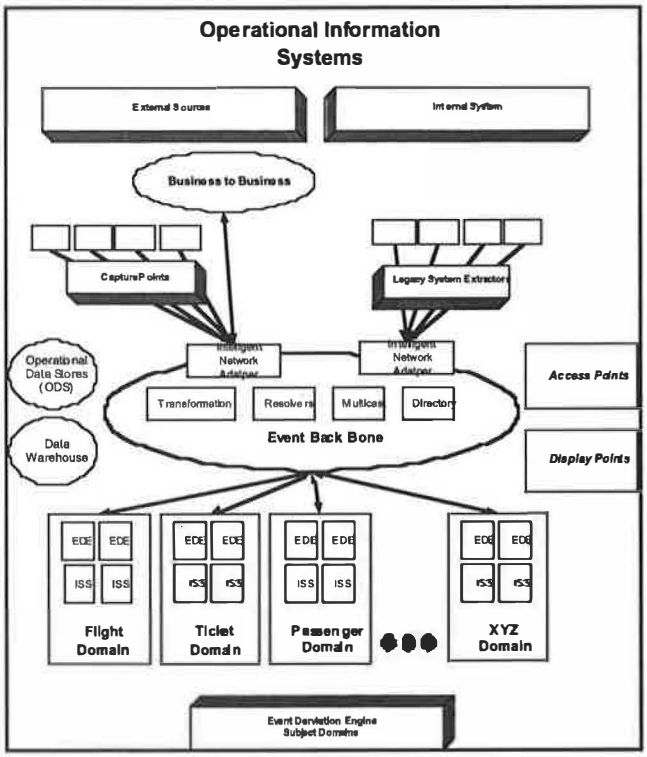


Figure 1 Systems Model

Intelligent Network is an IP based network that is embellished with strategically located resolvers and brokers to support inter-application communications. The application layer routing supports efficient and reliable message transportation.

Intelligent Network Adapters (INA) connect systems to the intelligent network. Delta's core legacy operational applications are implemented on the TPF operating system, which manages a loosely coupled cluster complex of IBM s/390s with a shared file system. Until recently, this operating system did not support a TCP stack and integration with IP networks was accomplished via gateways and custom protocols. As the TPF supported TCP stack matures, it will become compelling for some application interactions. However, a novel approach to this problem is being implemented by Delta. By using a hardware supported off-load engine that emulates the 3490 Tape interface, applications need not be modified to use the TCP interfaces. Applications simply continue to write and read to tape as they continue to assume a tape device.

This smart control unit also supports transformation to various contemporary message encodings, such as XML and additionally provides replication and brokering.

Operational Data Store (ODS) supplements the EDE. The ODS maintains a much larger operational window than that of the EDE. Additionally, the ODS accommodates alternative access styles such as complex analytical queries. The data store may also serve as a system of record for new operational objects that are not implemented in the legacy systems.

Initial State Service (IIS) is a mechanism, from which event based clients retrieve an initial view of information prior to receiving events that update that view. An example is a flight information display, where updates for flights may arrive at a client sparsely. That is, few events arrive over time. A passenger for a flight is interested in its current status. The initial view provides this current status in lieu of a status update event.

Access Points can both capture information and therefore, produce events, and also manipulate it. An important role of an AP is to permit the addition of new services, such as passenger paging upon flight arrival, dynamic pricing based on passenger profiles and current flight/airport status (e.g., availability of seats on competing flights), etc. These examples also demonstrate that APs may be connected to various output devices, such as pagers. Another AP is a baggage system for lost baggage. Passengers could register for baggage status events via a personal data assistant, which can be notified of ultimate arrival of the baggage. From these examples it is apparent that APs also vary, ranging from palmtops with wireless connections used by roving gate agents to the reservation-capable systems used by central airport agents.

Capture Points are any internal or external source of information. One example is an aircraft that emits positioning signals for capture in the operational service. The system's distributed capture points (CP) continuously emit events describing current status, using typed event records with unique instance IDs. CPs range from being low-end and ill-connected (e.g., wireless data entry devices used on the tarmac), to being high-end and well-connected, such as the customer-visible gate readers that scan boarding passes as passengers board, automating the boarding process. Consequently, the events produced by CPs also vary in complexity, one of the more complex events being an arrival event for a flight with a certain ID, such as a Surface Movement Advisory (SMA) at an airport; such events are contained in the FAA data feed.

The Operational Information System is composed of four fundamental processes: event acquisition, event consolidation, operational data storage, and derived event publishing.

3. Event Mining and Acquisition

The first of the four basic processes of the OIS is the acquisition of events from source systems. This includes the techniques for mining and tapping event sources, the ordering properties of transaction histories, as well as the publishing and transportation of this captured information.

The model used by Delta, as in other operational settings, is that of acquiring and replicating transaction histories to the Event Derivation Engine. Some of the specific information captured, generated, and transported in Delta's OIS includes flight, passenger, crew, situational, and environmental data. Some of these flows are produced by internal OLTP systems, such as flows that contain flight, passenger, and baggage information. Other flows are provided by external sources, such as FAA feeds, which provide radar-gathered positional flight data and weather feeds provided by a weather service.

3.1 Transaction Tapping

Transaction snooping and software agents are two basic techniques for tapping transaction systems. In either case, it is imperative to minimize the intrusion in the legacy systems. The core OLTP system was initially planned to processes several million well-behaved transactions per day. When tapping the legacy OLTP systems, existing service agreements must be maintained so that current users see no degradation in performance. Therefore, techniques for tapping must be minimally intrusive.

Transaction snooping is using a non-intrusive means of `grabbing' transactions as they occur. For example, modern OLTP systems incorporate sequential transaction logs for recovery purposes. With knowledge of the log format and the ability to view the log, transactions can be detected and acquired. The captured transactions can then be forwarded to a brokering engine for dissemination. That is, by utilizing memory-based table references, the

transactions can be decoded and reformatted for transmission to an OIS. This is straightforward for legacy applications that are altered infrequently, as the reference table must be updated with any transaction change. This technique is highly compelling, since hardware support can be used to snoop transactions as they are written to the logs.

An alternative technique is the utilization of software agents injected into the applications of an OLTP system. As non-intrusively as possible, these agents build records over the lifetime of some business transaction. They then fire triggers that generate appropriate events into a transport mechanism to make the data accessible to the new mid-tier OIS.

Both techniques are used at Delta, since many of the legacy TPF applications do not physically store the transaction boundaries in a transaction log for snooping. In order to capture the transaction context, the transactions must be gathered while they are occurring by a software agent. Upon commit, the transaction history is queued for I/O.

3.2 Transaction Ordering

The transaction histories must be complete histories of relevant interactions captured by the legacy system. Given such histories, the mid-tier OIS must be able to faithfully recreate and replay relevant operational state changes known to the legacy system and important to the mid-tier OIS.

Although some source systems provide consistent, reliable, and ordered messages that can be trivially internalized by an EDE, tapping some legacy transaction systems can result in an arbitrary re-ordering of the captured transaction histories.

The Intelligent Network Adapter used to integrate the TPF system with the OIS does not solely solve ordering anomalies that are introduced by the asynchronous I/O model used to transmit the captured transaction histories.

As an example of the ordering anomalies, consider the reservation system running on this loosely coupled cluster architecture. Specifically, when tapping this system's transactions for passenger status, we can acquire information about boarding status, seat assignment, customer status, etc.

To simplify the example, consider a system with 3 loosely coupled nodes, N1, N2, and N3. Assume that a transaction on a specific object instance can be arbitrarily routed to any node (this is a shared disk databases model, where any node can update an object such as a passenger record). For this example there are three transactions that update passenger record, "Jones". They are identified as T1_Jones, T2_Jones, and T3_Jones, which execute on N1, N2, and N3 respectively. Each transaction is properly serialized by the shared database and ordered by its occurrence. In this case we order T1_Jones happens before T2_Jones and T2_Jones happens before T3_Jones.

The problem arises as the captured transactions are asynchronously scheduled for I/O by the node on which the transaction occurred. This allows for transactions to enter the network not in order of their occurrence. That is T3_Jones can be sent before T2_Jones and T2_Jones can be sent before T1_Jones. If not re-ordered by the EDE, this results in an inconsistent view of the working set.

Synchronous coordination of the outbound transactions is detrimental to high throughput and scalability of the clustered complex. The asynchrony of the node processing can lead to non-deterministic delays. These delays result in large I/O queue depths that can ultimately result in back-pressure that affects existing service levels. That is, normal application processing can be affected.

Another scenario is the failure of a node, for which a transaction occurred. The transaction is not scheduled for I/O until the node is recovered. In this case a failure of N2, would result in a significant, possibly indefinite delay of T2_Jones. The EDE can't allow T3_Jones to execute, since this results in an inconsistency. If the node is not recovered in a reasonable time, the OIS must then re-synchronize with the legacy OLTP database for that instance "Jones".

Unfortunately, the legacy TPF applications do not encode the transaction boundaries in a transaction log and there is no corresponding unique transaction identifier. As demonstrated above, the ability to re-order a transaction-history is vital to the consistent reply of transaction histories in the EDE.

To account for the arbitrary re-ordering, Delta incorporates instance-based application sequencing to order

the captured transactions. An instance is an object, "Jones", that has been modified by a transaction. All transactions occurring on the passenger record, "Jones", are sequenced by a monotonically increasing sequence number.

This instance based sequence number allows the EDE to appropriately re-order the transaction histories. The instance-based sequencing technique has a profound advantage over a traditional unique transaction identifier. The concurrency potential is dictated at the instance level. That is, when a message for the instance, "Jones", is indefinitely delayed, all other instances can be consistently re-played in the EDE. Only the instance, "Jones", is required to be re-synchronized. This allows the EDE to achieve optimal levels of parallelism, by using an instance based concurrency controller.

Application instance sequencing is critical in the loosely coupled cluster since there are more opportunities for ordering anomalies by the cluster. Additionally, the relative frequency of updates to an instance is high therefore the probability for re-ordered transactions is high.

As an example of an intolerable inconsistency, consider gate agents utilizing a new application of the OIS infrastructure. By using real-time updated seat maps, agents have current knowledge of seat assignments. However, if inconsistencies were allowed, a passenger could show up with a valid boarding pass, however, information reflecting this may not be consistent at the gate. In fact, the passenger could be denied immediate boarding as he scans the boarding pass, which is rejected. Of course the passenger will be allowed to board after reconciliation, with the legacy system. However, this defeats the benefits of such a system to improve boarding times, and the overall customer experience.

3.3 Event Taxonomy

A challenge exists in that the information streaming from the loosely coupled systems (and/or from other sources, such as the FAA data feeds) is not delivered at the granularity useful to current or future applications.

Unfortunately, the resulting events produced by the legacy system do not individually contain the information needed by various business processing performed in the mid-tier EDE. To address this issue and to be able to handle diverse input streams to the EDE, we have developed the following characterization of events produced by external systems:

Discrete events are semantically meaningful to some OIS application. Upon receipt by the EDE, such can be immediately published.

Partial events implement state changes that in themselves are not useful to an application. Such events are directed to state engines, which will eventually produce a discrete event for some application. Partial events may be received from multiple sources (i.e., event channels) before causing a state change and therefore, a discrete event relevant to an application.

Incomplete events result from the ordering anomalies introduced by the clustered OLTP systems. As described previously, these events must be stalled until missing events arrive or the system deems this instance is not recoverable, resulting in re-synchronization processes.

Complex events are comprised of some combination of discrete and partial events. Such events are useful when applications require larger granularity activations than those resulting from discrete events.

These classes of events motivate a consolidation and correlation tier, where the Event Derivation Engine collects the event flows and derives events application-friendly events.

4. Event Derivation Engine

The second process of an OIS infrastructure is the correlation and consolidation of the tapped data from internal and external sources.

When information from internal and external capture-points is acquired and delivered to the OIS, the EDE exercises business rules to create new associations and representations of the information. For example, when the status of a flight changes, these state changes are delivered as events from capture points (e.g., the aircraft or the dispatcher) to the EDE. Here, the resulting updated status is internalized and represented in the current operational working set. With this working set defined, interfaces are provided for interested applications, which may request the current state of these new information representations and subscribe to the resulting state changes as events.

In practice, the event rates have already exceeded a non uniform distribution of over 12 million messages per day (Figure 2) from the internal and external sources that publish to the OIS infrastructure. This number is expected to rise significantly as more useful information is captured for event derivation.

Although this rate appears trivial, it is not amortized uniformly over the 24-hour period. Airlines incur high frequency peaks and transaction rates can double during holidays, fare wars, and strikes.

The EDE must maintain some subset of these flows as a base from which to derive application-friendly events. That is, information from external and internal sources does not typically match the expectations of consuming applications and must be correlated with other data to establish meaningful events. Additionally, initial states, described later, queried from the base.

Activity	Messages/Day
Flight Progress (TPF)	500,000
Flight Progress (FAA)	250,000
Passenger Information	3,500,000
Tickets	2,000,000
Inventory	5,000,000
Seats	500,000
Customer	100,000
Fares and Rules	200,000
Total	12,050,000

Figure 2 Message Rates From External Sources

For a perspective, as compared to data warehouses, which typically contain tremendous volumes of historical information, an OIS contains only the fundamental subset of information required to run day-to-day operations. Although the operational working set is a much smaller set, the aggregation of operational flows from external and internal sources can result in operational data stores of Terabytes in magnitude.

As previously stated, flow aggregation results in Terabyte-size databases. Maintaining these databases coupled with analytical processing on the data are two fundamental tasks of the mid-tier OIS. Other tasks include the acquisition, derivation, and publication of events with low latencies and in soft real time. Considering the demands of these tasks, an important observation is that the order of magnitude of the data from which application events are derived can be dramatically reduced, by focusing on precisely the data needed for near-term operational decisions and actions.

Therefore, we improve event throughput and latencies by defining a derivation subset, named the Derivation Working Set (DWS), which is of much smaller scale. The DWS contains the minimal amount of information needed to derive the events required by OIS applications. Performance of data storage and access for event derivation is improved substantially because this working set can be implemented as a main-memory database that is optimally organized to accommodate event derivation and initial state queries.

The DWS scoped via a window scheme, where content is rolled in and out of the DWS based on relevance. Specifically, in this set is kept all state of 'current interest', so that it is rapidly accessible to relevant business logic. For instance, data about a flight's departure is kept in the DWS until the flight has arrived, whereupon 'business logic' adds to the DWS that the information that a certain flight leg has been completed. The lifetime, or window, of the information contained in the DWS is based on business operations for a specific business domain. For example, years of experience in dealing with flight information resulted in the identification of a window of flight data and behavior for some number of days (n) in the past to some number (m) days in the future. Lifetimes vary across business domains, and they may also be dynamic, such as lifetimes based on event arrivals. For instance, as a flight leaves a gate and begins taxiing, the boarding process for that flight is no longer relevant and may be flushed to the Operational Data Store (ODS) and possibly, to the data warehouse.

The EDE is the primary data provider and consumer for additional services associated with the operational subsystem, such as Internet-based reservations and flight information services, the reservation system used by external systems in a business to business model. Finally, the EDE also directly distributes events to display points, such as flight displays in airports, resulting in the need for high scalability (in terms of numbers of displays) for some of the event output streams emanating from the EDE.

EDE Processing Model

1. Transaction History (TH) arrives from source system.
2. Durably store the TH.

3. De-marshal TH.
4. Exercise concurrency control rules
5. Exercise internalization business rules.
6. Release TH
7. Represent TH in DWS
8. Derive corresponding application event.
9. Publish event.

In general, the role of the EDE is to create meaningful global states from event streams that provide limited ordering guarantees.

6. Operational Data Store

The third processing component of the OIS is the ODS. The ODS window is much larger than that of the EDE and is typically implemented with traditional relational databases. Operational Decision Support applications as well as Data Mining applications use the ODS to execute analytical queries in the ODS environment where they do not compete with the real time event derivations performed by the EDE. Additionally, the ODS serves as a staging area for populating the DWS. For example, passengers can book seats on flights in the distant future. This information is considered `operational' by Delta. However, it is outside of the DWS window for the passenger domain. The magnitude of this type of future information is quite large, so it is staged in the ODS until it falls within the DWS window.

The DWS support the inter-operation of event flows with the EDE and the ODS, in which, the ODS and EDE collaborate to provide traditional transaction processing to this new base of operational information, without impeading the requirements for low latency events.

7. Derived Event Publishing

The fourth processing component of an OIS is the dissemination of events derived by the EDE to its large numbers of subscribers (e.g., airport displays). In fact an existing deployment already exceeds 10,000 workstations, which must display flight status information.

The highest profile service of the OIS infrastructure is the support of soft real-time delivery of event information to subscribing clients. Real-time event applications are the impetus for re-thinking business processes and revolutionizing the operations of the airline. The benefits are profound. Consider an example where gate agents are provided with heads-up displays that present a current view of relevant flight information, including seat maps for the flights they are working. The traditional request/reply strategy is limiting as agents spent their time working at the computer terminal, typically typing requests to answer basic customer questions. The heads-up displays allow both customers and agents to be informed releasing the agents to spend their time responding to more difficult issues, including facilitating the boarding process.

To achieve low latencies and high scalability, the relaxation of the reliability of event transmission is based on application characteristics. While some applications require stringent guarantees, others can operate successfully under relaxed rules, which we term the `reliability spectrum'. To exploit this spectrum the use of a hybrid sender and receiver-initiated multicast protocol can provided dramatic improvements in the latency and communications scaleabilty of an EDE.

Initial states must be obtained for any event-based application that requires the current value of a set of information to begin operation. An application that demonstrates this requirement is Flight Information Display Systems (FIDS), for which there are many receivers (e.g., the large number of airport displays) that join and leave this service in a periodic fashion. A FIDS display requires initial values for presentation and subsequently, adjusts these states as flight changes are received. For FIDS, the determination of initial state is as simple as capturing a small set of initial data, for others this may require making a copy of the entire operational working set.

8. Experiences

The overall architecture of the OIS components has evolved over time as the scaleability and availability requirements have changed. Initially the system was a proof of concept that acquired immediate success and was deployed well-beyond its designed capacity. The currently deployed system has been refined to meet the scaleability and availability requirements.

The initial EDE design used a commercial relational database to internalize the transaction histories and represent the operational working set. The original intent was to enable fast, flexible queries, along with low latency event distribution. However, as the operational working

sets grew to Terabyte magnitudes, we quickly realized the competition between maintaining large databases and fast event derivation from this database. Through our experience with this deployed architecture, we realized disk-resident relational data offered insufficient performance to solely handle all of the work required of the OIS infrastructure. Not only must the OIS process the variable peeks of the 12 million source messages per day, the OIS must additionally derive at least that many application friendly/discrete events to an initial deployment of 10,000 workstations. The desired number of workstations is expected to grow dramatically in the near future. Additionally, the explosion of initial state queries occur as workstations dynamically join/subscribe, which require initial states. For FIDS applications this initial state, which results in a XML result set of 5 MB places a tremendous load on the system. In an unlikely but worst case scenario, all current 10,000 workstations could come on-line at once requiring 10,000 queries.

This situation is exacerbated further by the existence of additional external systems, including passenger-booking traffic via the Internet. This will result in the existence of many more information flows and resulting analysis tasks in the future, ranging from `small' flows like automatic passenger paging services, to multimedia flows.

Delta immediately discontinued supporting the important feature of analytical queries from the OIS and began maintaining a scaled down in-memory representation of the working set. The relational database image was used for recovering this scoped state upon failures. Unfortunately, if failures in this system occur frequently, a business could face significant downtime. The time to replace the working cache from the several tera-byte RDBMS is on the order of 45 minutes.

This large volume operational working set motivated the partitioning of the set of information required for deriving events, the DWS, and the Operational Data Store. The ODS is organized to handle ad-hoc analytical queries, while the EDE derives events with lower latencies.

Additionally, client connectivity in the existing system, is based on a hierarchical fan out based on TCP socket concentrators. Delta has identified that this approach introduces un-necessary moving parts and adds latency as events traverse the hops.

Delta's experience and requirements in developing a commercially deployed OIS infrastructure has motivated our academic research in this space. The current scaleability challenges coupled with future scaleability projections stimulate a clean slate approach to researching more optimal architectures for an OIS.

9. Research Goals

The Operational Information System has a mission-critical dependence upon information that is acquired, processed, transported, and delivered with well-defined quality of service properties. The intent is to attain competitive advantages in decision-making, customer care, and to react in a timely manner to changes in current state. We must manage the processing and communications performed by multiple components of a large-scale distributed application. This application consists of large number of complex information flows, where operations applied to these flows have the purpose of extracting `useful' data from them. Information extraction must be performed under constraints like timeliness and continuous availability. Not meeting these constraints results in costs incurred by the organization.

In this section, we describe the research opportunities and academic interest in components of the OIS infrastructure.

9.1.1 EDE Scaleability

The EDE must be highly available and scaleable while also employing rapidly evolveable, plug-and-play hardware and software infrastructures, so that new services are easily added, data formats changed or updated, and additional streams and clients supported.

An opportunity to improve scalability of the EDE is parallelism of the transaction histories. Specifically, transaction histories from TPF can be executed in the EDE under the assumption of causality, which maximizes the parallelism in the stream.

An additional enhancement of the EDE is one that also replicates the DWS itself, to attain high levels of availability and scalability. We are designing solutions based on event mirroring and/or hot standbys, again using additional cluster computing engines.

9.1.2 Low Latency Events

One issue is the latency of outbound events, generated

in response to the receipt of new input events by the EDE and subsequent state updates. High latency for such events has strong effects on operational capabilities and on the customer experience, the latter exemplified by Delta's ability to rapidly display flight and gate change information. Latency is affected by both event transport and event processing overheads. We first consider processing overheads. While these overheads are reduced by using an in-memory representation of the EDE' operational working set, there exist additional latency and bandwidth issues caused by verbose event representations and the unnecessary copying of events in interactions with the event-based communication infrastructure.

We are addressing these issues as performance opportunities by incorporating portable binary input/output (PBIO) encodings of event transfer formats along with "Just In Time XML" [2],[3],[4]. Effectively, by organizing the in-memory representation of the DWS to more closely match the application-friendly/discrete events, the PBIO technique allows the EDE to simply drop the memory image on the wire. This dramatically reduces buffer coping in the sender, since marshaling to an intermediate representation such as XDR or XML is not necessary. To deal with sender memory packing and endianness, the receiver has pre-cached meta-information that describes the memory structures that it receives as events from the sender. The receiver re-constructs the data structure in its native packing and endianness format and additionally translate the message into a contemporary format such as XML. This process occurs under the PBIO library and is abstracted from the receiver's application code via some middleware package.

9.1.3 Communications Scaleability

Many applications can operate successfully in the presence of message loss and can take advantage of relaxed reliability protocols. This characteristic does not imply that the applications must be concerned with inconsistent views of the data. This characteristic means there are natural alternative means to ensure application information integrity. What is fundamentally required in this scenario, is the ability to detect event loss and the ability to re-synchronize a client application upon detection of message loss. Such is the case of the FIDS application of the OIS. If message loss occurs the FIDS client can re-synchronize by requesting an initial state and begin receiving events that update that state.

The performance/reliability tradeoffs of receiver- vs. sender-initiated multicast protocols are well known, offering stronger vs. weaker reliability vs. throughput, respectively [6]. For our environment, we have developed and are now evaluating a hybrid approach, in which attributes of both types of protocols are used to achieve a compromise for required reliability with high throughput. In this protocol, the receiver is responsible for lost message detection via sequence number analysis, and the sender buffers messages to accommodate retransmission requests. Periodically, the receiver issues a consolidated ACK for some sequence of messages, such that the sender can purge buffers.

To account for the implosion of these consolidated ACKS and NAKS by many receivers, NAK and ACK concentrators serves as a representative for some number of assigned receivers [5]. A sender is initiated and waits for a specified number of concentrators to join as representatives of the receivers. Next, the concentrators are launched, where they wait for a specified number of receivers to join their respective concentrator. When all expected receivers join the concentrator, the concentrator joins the sender and message flow begins. All ACKS and NAKS are forwarded via the concentrator.

9.1.4 Bandwidth Conscious Initial States

Some applications of the OIS require initial states on the order of 3 to 5 MB (non-compressed) in size. In situations where large numbers of display points simultaneously join the OIS event flows, this can have dramatic implications on the server load as well as bandwidth consumption. Since, display points are typically connected to the OIS via a WAN link on the order of 6 Mbs, the bandwidth alone allows 15 initial states per minute. Incorporating 90% compression increases the rate to 150 per minute. However, compression induces additional overhead and 150 initial states per minute remains far from sufficient.

Delta's current OIS utilizes a hierarchically organized distributed server caching architecture for handling initial state load. Since caching and the computation and re-computation of initial states can have significant performance effects, especially for the highly available, replicated EDE, we are pursuing a scaleable and bandwidth conscious solution to this problem. Potential techniques include a combination of techniques, such as, PBIO, Forward Error Correction, Local Recovery, Periodic State Multicasting, and Distributed Initial State Servers.

10. Simulation

To simulate a possible future commercial deployment, we define our target simulation of an OIS that can support on the order of 100,000 capture points, with larger future systems having up to 1,000,000 capture points. We vary the event types, from 100 for a basic system, to 5,000 for a complex system. We model a basic, mid-end AP system as supporting 5 output devices and 5 input devices per AP, a complex system supporting an additional 500 output devices (e.g., passenger pagers) and able to deal with inputs from other sources (e.g., direct conversations with other APs). Consequently, the number of event types handled per AP varies from 50 to 500. Initially, we assume the current transaction rate of the commercial system. However, our research will experiment with much more demanding event systems, including those that emit continuous events, such as real-time aircraft status including pitch, yaw, and thrust, which would be tapped directly from an on-board control bus.

11. Conclusions, Status and Future Work

We have described the research and commercial opportunities presented by operational information systems, and their strategic importance to Delta Air Lines. An interesting approach to building new systems pursued by Delta is to tap its legacy operational systems, then reproduce desired images of operational information for new, mid-tier operational information systems (OIS). The idea is to create additional systems on which new business applications can be developed, without jeopardizing already existing systems and their operation.

Issues that arise for operational information systems include: (1) the efficient capture, transport, and delivery of events carrying operational data, in the wide-area environment in which Delta must operate, (2) the online derivation of consistent views of operational data, based on which various applications in the OIS may be run, including dealing with incomplete operational data and with diverse orderings and timing for event receipt, and (3) creating highly available OIS components.

We are continuing our research into the efficient, scaleable, and low latency processing and distribution of events, by evolving our communication/event infrastructures and OIS event processing and storage engines. In addition, we will pursue scaleable methods for wide area event distribution and collection. Finally, we are investigating the creation of highly available COTS cluster machines for the new mid-tier OIS' operational data engines and stores, so that these systems can offer the availability and reliability now offered by the existing legacy infrastructure.

12. Acknowledgments

We thank Mark Whitney, Scott Gosline, Deborah Callahan, and Rick Lawhorn whose vision, dedication, and perseverance have resulted in the novel OIS described here. Their work has already been recognized by receipt of the "Smithsonian Award". Thanks also to other members of the Delta team including Ron Byrd, Dean Compton, Bret Martin, and Mike Manatrizio for supporting our research. Patrick Widener and Mustaq Ahamad have also participated in our research and simulation activities.

References

[1] M. Ahamad. Causal memory: defnitions, implementation, and programming. Distributed Computing, pages 37{49, 1995.

[2] Greg Eisenhauer. Portable self-describing binary data streams. Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, 1994. http://www.cc.gatech.edu/tech reports.

[3] Greg Eisenhauer, Fabian Bustamente, and Karsten Schwan. A middleware toolkit for client-initiated service specialization. In Submitted to PODC Middleware Symposium, 2000.

[4] Fabian Bustamente, Greg Eisenhauer, Karsten Schwan and Patrick Widener, "Efficient Wire Formats for High Performance Computing", To appear at SC'2000

[5] J. C. Lin and S. Paul. RMTP: A reliable multicast transport protocol. pages 1414-1424, March 1996.

[6] Sridhar Pingali, Don Towsley, and James F. Kurose. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. In Proceedings of 1994 conference on Measurement and modeling of computer systems, pages 221-230, 1994.

Experiences in Measuring the Reliability of a Cache-Based Storage System

Dan Lambright
EMC
Dlambrig@emc.com

Abstract

We present our experiences in benchmarking the reliability of the cache component of a storage system in a development environment. The reliability metrics we measured are availability from the standpoint of the host and maintainability from the standpoint of the system operator. We created errors using software fault injection, and measured their impact using a combination of performance measurement techniques and the rehearsal of maintenance procedures. This paper gives three case studies. The first two describe experiments that recreate very specific breakdowns in the software logic, and the third describes an experiment simulating a memory hardware failure that creates unpredictable effects. We found that, taken together, these various techniques gave us a useful picture of how well our cache management software tolerated faults.

1. Introduction

Measuring the reliability of software, a relatively unexplored topic within the systems community, has recently been dubbed one of the major challenges facing computer scientists who come from that background [1].

From the standpoint of industry, the positive feeedback that reliability measurements bring is clear. Doing so helps determine whether one version of software is more reliable then a different version on the same system. It can alert developers to where attention should be focused, in particular whether valuable time should be spent writing a complex or simple solution. Creating errors also acts as a trial run for the recovery process employed by the system operator or customer service engineer, testing the robustness and effectiveness of diagnosis utilities for locating and fixing the problem.

In this paper, we discuss our experiences in measuring the reliability of the cache component of a modern disk array. We did not attempt to benchmark the entire system, choosing instead to focus on the cache because it lies at the heart of the architecture of the product. Errors frequently appear in cache before they appear on the disk [6]. The techniques we describe are by no means revolutionary. However, our goal is to demonstrate the usefulness of the techniques, and to give some insights into where future research could be directed.

In the product we examined, the error detection and recovery software of the cache subsystem is quite sophisticated. The design has no single point of failure and a great deal of resiliency in its structure. This work is intended to work torwards developing techniques to check if the hardware and software satisfies that design goal. Additionally, we desire a more explicit mechanism for determining how the software's effectiveness improves as it evolves. Presently, deriving that notion can be done by studying a dispersed set of statistics ranging from the binary results generated from regression testing, to reports summarizing the errors generated at customer sites. The desire for reliability benchmarks stems from a need to have a more immediate, reproducible, source of information.

The product we examined had software and hardware specifications readily available and modifiable for the purpose of our tests. This allowed us to employ software fault injection, a widely used technique, to generate faults [2][9][12].

The fault's impact on availability was in part measured with tools and techniques used by our performance measurement group. The fault's impact on maintainability was assessed by rehearsing the formal procedures a system operator would have taken in case of the fault, and noting what happened. We created faults to test both narrow, targeted points in the software logic ("targeted faults") and at broader problems ("untargeted faults"). The former technique was useful for stressing important algorithms in a very specific way. Realistically, such tests could not be custom designed for all the algorithms on the machine, and only provided reliability information on a small portion of code. For the system in its entirety, we turned to the later technique. Untargeted faults provide a reliability metric at the granularity of the system rather then the algorithm. By using both techniques selectively, we were able to get a more complete picture of reliability.

This paper is organized as follows: Section 2 characterizes the systems we benchmark. Section 3 discusses our methodology for generating faults and measurements. Section 4 contains studies showing how "targeted" faults test a narrow (but important) aspect of the system's reliability. We found faults that acted randomly in a scattershot means provide better statistics on overall system robustness. Section 5 discusses those experiences. Section 6 explores some ideas relating to the integration of reliability benchmarking into software labs and future directions in research. Section 7 reviews related research, and section 8 concludes.

2. The Storage Device and Cache Subsystem

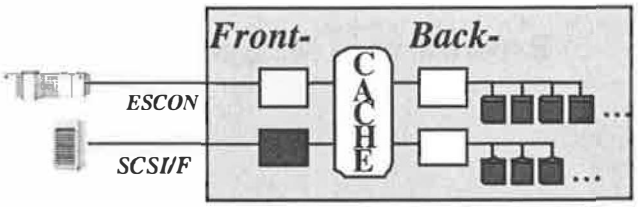
In this section, we give a very high level description of the architecture of the system we test and the role the cache plays in the system. We then describe in more detail how faults which appear in the cache are detected and categorized.

The product is a large disk array. The storage may be accessed over multiple I/O channels, which may be connected to varying types of hosts (e.g. MVS, UNIX), that consequently use varying protocols (e.g. ESCON, SCSI). A set of processors that we collectively call the "front-end" are devoted to interpreting the protocols and communications from the host systems. Another set of processors (the "back-end") operates the disk drives. It consists of a set of CPUs that are divided between the disks according to configuration. The CPUs may communicate with each other via one or more networks, or a centralized memory, or both.

A large cache in the product serves as a holding place for data between the front and back-end. The cache is accessed from the CPUs over redundant, high-speed backplanes. On write operations, the front end CPUs transfer data from the channel to the cache, and the back end CPUs that control the target disks transfer data onto storage. On read operations, the path flows in reverse. Data in cache that has not yet been written to disk is "dirty". Data is removed from cache using a variant of the LRU algorithm.

The system cache size may change dynamically. For example, if a given threshold of errors is detected, the cache is dynamically fenced off so that it no longer can be used. Additionally, the user may add, remove, or replace units of memory with minimal impact on availability to the user.

High Level Block Diagram of System



Errors in the cache are grouped into two categories: hardware and software. A software fault is a "bug" that resides in the code running on the front or back-end CPUs. A hardware fault consists of a number of bits which no longer function properly during read or write accesses within a given cache line. To be categorized as a hardware error, the number of faulty bits must exceed that which is correctable by the hardware's EDAC (error detection and correction) logic. Hardware faults in the cache may also be manifested in faulty components (e.g. the front end or backplane malfunctions). There is redundancy at each level of the hardware to mitigate the impact of such problems.

Cache errors are further categorized by the impact they have on the user. For example, availability faults hinder performance but do not corrupt data. A hardware error that corrupts meta-data related to the LRU algorithm might affect optimal replacement strategy, but the user would still be able to load data at some reduced rate or response time.

Other metrics used to describe cache errors include latency and burstiness. Latency quantifies how long ago the fault occurred. Clearly, the shorter the latency, the easier it is to detect root cause, especially if the amount of space devoted to logs is limited. Burstiness describes what errors cluster together. A cache error may propagate to other errors on the disk or channel controller, or may be a symptom of a problem originating in those subsystems.

The system tracks errors using logs that preserve the context of the fault (e.g. stack trace, counters) for debugging. Errors are detected in the cache by background scrubbing tasks and during the I/O operations. The frequency of running the scrubbing software (which impacts error detection latency) must be balanced with providing enough time for processing host requests.

The cache has a set of diagnostic and development tools used to monitor aspects of the subsystem, including the state of the LRU algorithm, host utilization, meta-data associated with the cache (e.g. software locks), and error

detection and recovery. The proper functioning of these diagnostic capabilities, even in the presense of severe faults, is important.

In summary, we are verifying the reliability of one of the components of a highly available system. We note that the system is made up of many different redundant subsystems, each of which could be analyzed separately. It would be useful to analyze the entire system as a whole, but the variety of hardware would make this a more difficult project.

3. Methodology

In this section, we describe the test configuration for our reliability measurements. Because of limited resources we had to trade off accuracy in our measurements for expediency. Nevertheless, the configuration was successful in helping us reach conclusions. We then discuss the metrics we use to measure availability and maintainability. For maintainability, our measurements were based in part on the subjective analysis of human beings. This is because of the complexity involved in developing automated measurement techniques.

To quantify availability, we adapted the general methodology for availability benchmarking to our environment [1]. Essentially, this procedure works by injecting one or more faults into the system while measuring a Quality of Service (QOS) metric. In our case, we wished to know the fault's impact on overall response time and throughput. These metrics appealed to us because they were already well-established in our vocabulary (for describing performance), and we had mature techniques and instruments, as well as seasoned in-house specialists, to do the measurements. Additionally, we knew of several applications (routinely used for our functionality tests), that were sensitive to unexpected deviations in those metrics.

Our workload generator consisted of a dedicated MVS mainframe running scripts to generate I/O. The storage system was connected to the host over 12 ESCON channels. The storage system had 8 GB of cache. There were 96 physical hard drives on the machine, each with a capacity of 18GB. The drives were partitioned into 288 "logical volumes," which were the "disks" visible to the host. We performed no other I/O or special applications on the storage subsystem.

For each test we ran a mix of 25% writes and 75% reads on randomly chosen blocks on the disks. This was a crude approximation of customer behavior, and ideally, we would prefer an I/O stream that predicted the impact on customers by mirroring actual user behavior. However, the behavior of users varies greatly from application to application. Creating a single profile representing the "average user" is beyond the scope of this experiment. In performance testing, different tests are run to test different classes of common I/O profiles (e.g. online transactions, sequential write). This will most likely be our course of action for future work.

We generated I/O requests from the host at a single rate, which was at a relatively low level compared to the maximum the system could handle. We took this approach in order to approximate the level of I/O of a typical system, rather then the level of an "envelope test", which might never be seen outside of performance benchmarking labs. Our measurement software, originally designed for performance testing, recorded the response time and I/O throughput once every minute. Our availability measuring tools were not accurate, but this was acceptable, as we were more interested in understanding the fault's impact then getting a high degree of precision.

The effect on maintainability was evaluated by manually simulating the corrective actions that the customer service engineer would take in the case of a fault. Those actions were known by following the instructions corresponding to the error in a knowledge database used by customer engineers, as well as interviewing them in person. When no solution in the knowledge database matched a fault, we determined what a customer engineer would do by conducting interviews and following our own judgement.

We considered writing an automated script to perform maintenance functions (derived by viewing logs of error recovery situations that had occurred in the past), but concluded this method was not helpful. A typical "solution" to a problem, as carried out by the system operator, involves a sequence of decisions, which are manifested by entering different diagnostic commands depending on the state of the machine at the time. A log only contains a sequence of commands for one unique situation. Simply "playing back" the same sequence of commands during fault injection could be inappropriate depending on the state of the machine at the time of the fault.

For example, suppose a fault in the cache was the type that propagated to another subsystem, such as the back-end CPU. The technician's first job would be to work on

those problems, which would include checking the integrity of the disk. If the disk was a member of a RAID group, the data may be in several intermediate states. A script that accounted for the myriad possible states the machine could be in would be extremely complex and potentially error prone.

Maintainability was quantified using three parameters. Each parameter was rated on a scale of increasing quality: low, medium, and high. We call the first parameter effectiveness of diagnosis tools. To obtain it we noted the correct existence and operation of diagnosis utilities and error information. For example, in some cases we found that the faults we created did not have utilities to diagnose the problem. To obtain information the technician would be forced to have a relatively deep understanding of the code and dump raw memory and interpret it. In such cases the effectiveness of the tools would be judged to be of lower quality.

The second parameter was simplicity of solution. For example, a simple recovery would be to invoke a software correction utility, and a more difficult recovery would be to physically replace a component that failed, or upgrade the code. It is almost always preferable to employ the former solution.

Our third parameter, robustness of diagnosis tools, quantifies the correct functionality of management interface software in response to severe problems in the storage system; severe problems in the latter should not cascade into the former. Systems under intense stress should still be capable of interpreting such utilities and maintaining logs, otherwise the problem would persist, and there would be no corrective option other than shutting down the system.

Lastly, the degree of severity of the generated fault was tunable. This allows the tester to gradually increase the severity until the effect becomes noticeable, or the system completely shuts down. This "point of no return" is a useful data-point, even if it is completely unrealistic and would never happen under real circumstances.

In addition to availability and maintainability metrics, we recorded: (1) how long it took for the problem to be detected by the system, (2) whether it was self correcting, (3) whether it was streaming (i.e. recurring repeatedly).

To sumarize, we had little trouble finding existing techniques to measure availability, but found that obtaining measurements to measure maintainability had to be invented. A criticism of our maintenance tests may be that because our opinions on the "quality" of maintainability are subjective, they are subject to controversy. But there are inherent difficulties in automating maintenance tests. Finding practical solutions to those problems may be an interesting area of research.

4. Targeted Fault Injection

In this section, we describe the "targeted" tests we performed to measure the resilience of particular code modules and specific recovery paths. Our first case study examines a situation where an important data structure is not synchronized between the different CPU's, and our second describes the effect of a rogue CPU that holds a shared software lock for unacceptably long periods.

4.1 Case Study 1: Unsynchronized Memory Maps

The first fault was a synchronization error. We wanted to test the behavior of the system when the different CPUs in the front and back end did not see the identical map representing which portions of the cache were available or fenced. In effect, some number of CPUs would believe that more memory existed than others. We believed such a problem would manifest itself in transient cache errors, but depending on the severity of the problem the number of errors may impact the overall availability of the system.

We wrote fault injection software to purposely break the synchronization of the memory maps. We adjusted the severity by changing the number of out-of-sync CPUs, and the size of cache (expressed in 32 MB chunks) disagreed upon. We attempted to break synchronization between both the front and back end CPUs to learn what difference that made in availability. We hypothesized that if the front-end CPU had more memory visible it would fill it with data on write operations, and the backend CPUs would then post an error when the dirty track was detected. Conversely, if the backend CPU had more memory, it would fill it on read operations, and the error would be posted immediately as the front-end CPU responded to the host.

For faults of low severity, we found no measurable effect on performance no mater which CPUs were out of sync. However, when we increased the number of out-of-sync CPUs to half those in the system, performance was noticeably impacted. Figures 1 and 2 illustrate the difference in impact between low and high serverity tests when backend CPUs were modified to have more memory visable. The respecting figures show the system's response

Figure 1: I/ O Rate for 1 Gig Out of Sync on 1 backend CPU

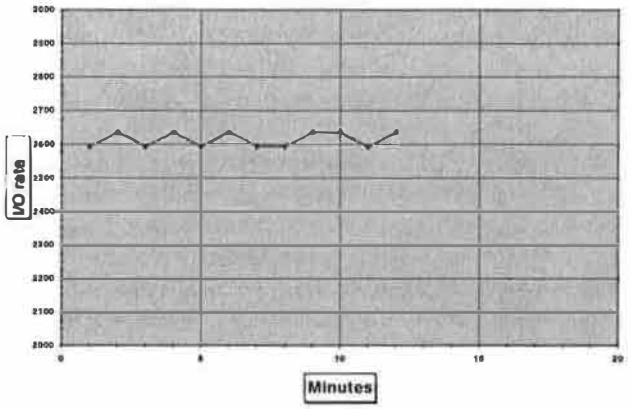
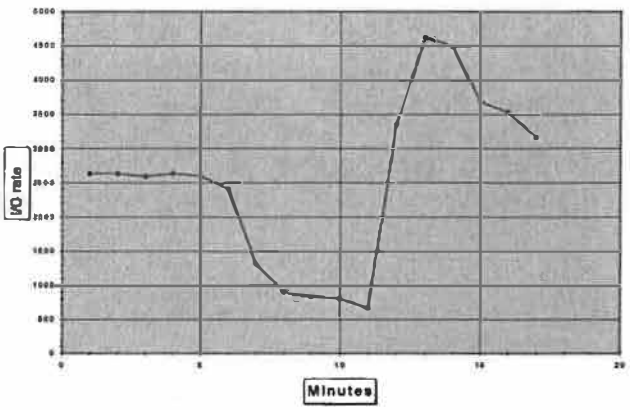


Figure 2: I/ o Rate for 4 Gig Out of Sync; all backend CPUs



time over a 20 minute interval (displayed on the x and y axi, respectively), in which a synchronization fault was injected and repaired. In the low severity test, no effect on response time was noticed. In the high severity test, the fault was injected at minute 5 and corrected at minute 10. At minute 12, the response time jumped as the system apeared to catch up with requests that had been delayed.

The degredation in response time occurred because one part of the system would attempt to access disabled memory, generating an error. Each time this happened there was a small delay to report the error. The greater the severity the more the delays aggregated, hence this recovery period grew longer as the severity increased. We did not see cascading errors at low severity, but we did at high severity.

We found that at high severity, after we fixed the problem (by re-synchronizing the memory maps using diagnostic utilities), there was a brief period of continued performance degradation, the cause of which we are investigating. We also found that when we repeated the same high severity test on front-end CPUs, the impact on throughput was somewhat greater, which was in accord with our hypothposis that front-end CPUs are more vulnerable to this type of problem.

Maintainability in the synchronization case was attainable. The diagnostic procedure and problem discovery process was to manually check the memory map on each CPU, compare that with others, then disable the memory banks until all CPUs saw the same memory map. The transient errors did not affect management software functionality even at the greatest severity. We therefore rated robustness to be of high quality. However, the diagnostic utilities only showed the memory maps for individual CPUs, rather then all of them at a time, and they did detect the problem (i.e. they did not verify identical memory maps on each CPU). We therefore rated the diagnostic effectiveness to be of medium quality.

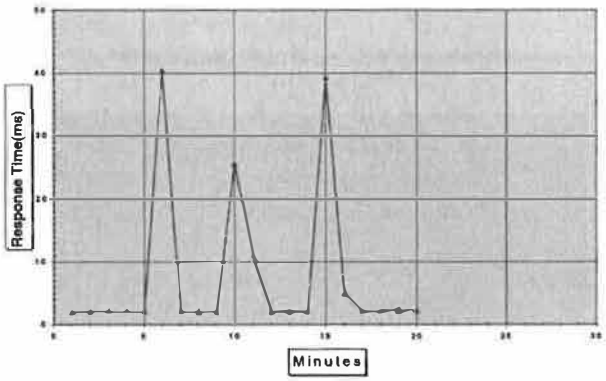
One concern was that to resynchronize the memory maps the CPU had to undergo a subset of the initialize microcode load (IML) processes. While this was a strightforward operation, it was time consuming enough to delay a small number of I/O completions. Simplicity of solution was therefore graded medium.

4.2 Case Study 2: Broken Locks

Our second test was to force improper functionality of a cache software lock. The purpose of the lock is to protect meta-data related to the LRU replacement algorithm. Our fault injector simulated a rogue CPU that had gone into a loop in which it repeatedly took and held the cache lock. When this occurrs beyond an expected amount it prevents other CPUs from accessing that meta-data and can delay I/O completion. We tuned the severity of this fault by increasing the frequency and length of time the lock was taken over a 20 minute period (e.g. at low severity the lock would be taken once and held for a period of 50 microseconds, and at high severity the lock would be taken 10 times and each time held for a period of 5 seconds).

As we expected, availability was impacted by this fault. The longer the lock was held the greater the host impact. Beyond a particular point the host timed out on the I/O, which we recorded as a cascading error. We also found that, as in the first test, there was a recovery period during which performance was still impacted. Figure 3 shows the results from one test of high severity. In the graph, the lock was taken by the rogue CPU 3 times, and held for a duration of one minute at each instance. The

Figure 3: Impact on Response Time with Misused Lock (High Severity)



graphs shows a 30 minute period (represented on the x axis) versus the system's response time durring that period. At minutes 6, 10, and 15 the rogue CPU took the lock and held it for a period of 20 seconds.

At low severity (lock held for less then one second), the response time was not impacted. In between the two extremes of high and low severity, response time varied. We were able to gradually increase the severity to find where the effect began to become noticeable. Beyond a certain point a CPU waiting for the lock would assume the rogue CPU had malfunctioned and so would take the lock by force, thus the problem was self correcting. However, in our high severity tests the rogue CPU would continually re-aquire the lock.

We discovered several management utilities used the lock in order to function and their capabilities degraded when the fault was severe. This led us to grade robustness to be medium. We suggested a "force" option to these utilities to bypass taking the lock. In doing so, the acuracy of the utility would be impacted (because exclusive control of the meta-data would not be held). This would normally be an acceptable tradeoff, however, because in a real-time debugging scenario obtaining timely information is more important then perfect accuracy.

We found this problem difficult to diagnose. When we presented it to a customer service engineer (without telling him of our experiment), it took him a longer time to determine the root cause of the problem, relative to the first experiment. At the highest severity levels (an endless loop repeatedly taking the lock), the only course of action is to reset the CPU to terminate our fault injecting program. We graded simplicity of the solution to be low.

Because the available management utilities were relatively obscure and cryptic (they were used primarily by developers as opposed to system operators), we graded their effectiveness to be medium. We recommended adding a new counter into the lock mechanism in a prominent display utility. It would be displayed in a red if the lock was taken more than some number of times within an interval.

4.3 Conclusions from Case Studies 1 and 2

With the knowledge gained by these two tests we could more easily determine whether devoting development time towards early detection of these faults would be useful. In the first case, a complex mechanism to ensure synchronization between the CPUs could be designed. But because the complexity would be high and the impact of the bug was relatively low (and simple to detect were it to actually occur), the arguments to stick with a simpler recovery mechanism won out. The second problem (greedy locking) was more severe, but because of the unlikelyhood of its actual occurance we confined our recommendations to improvements to management utilities.

5. Case Study 3: Untargeted Fault Injection

In this section, we discuss our tests for simulating hardware faults in the cache memory. The goal of these "untargeted" tests was to verify that in the presence of these conditions the system would generate a brief sequence of transient errors before fencing off the offending memory, switching to a write-through mode (if not in one already), and alerting the operator.

A hardware fault manifests itself when the software fails to perform a read or write operation to the cache. Whenever this occurs, the software should follow a recovery path rather than assume the operation succeeded. Because the cache is accessed from many points in the system, there are many recovery paths to test. Rather than laboriously test each one (the targeted aproach), we created faults randomly and attempted to get a statistical sense of the correctness of the recovery paths.

The fault injection software works by inverting enough bits in a cache word to defeat the error correction code (ECC). On our system there is a mechanism to disable ECC generation. We disabled ECC generation, wrote random data into the cache word, then re-enabled ECC generation. At that point the ECC no longer corresponds to the data and so will flag an error when it is accessed. The location affected in the cache was adjustable to be LRU

Maintenance Tests Results			
Measurement	Case Study 1	Case Study 2	Case Study 3
Coverage	Targeted	Targeted	Untargeted
Diagnosis Effectiveness	Medium	Medium	High
Diagnosis Robustness	High	Medium	Medium
Solution Simplicity	Medium	Low	High

algorithm meta-data or overall system state. Although certain regions of memory (such as system timers) are more likely to be accessed quickly by a CPU (thereby triggering the fault) than others, all memory errors will eventually be found by background memory scrubbing processes.

The fault injection software's severity was adjusted by increasing the number of faults created. In our experiments, we increased the number of memory faults in granularities of 2, 10, 100, 200, 400, 1000, and 250000000. We did not benchmark availability using the performance techniques of the previous section because the act of fencing memory directly affects performance because the size of the cache shrinks. An availability benchmark would have to distinguish that expected degradation from the unexpected consequences of software errors. We left the project of learning those interrelations to future work.

At low severity (granularities of 1,2,10), our tests showed some diagnostic utilities functioned marginally slower, and a limited number of transient errors (affecting neither availability nor data) were generated. When we increased the severity to between 200 and 1000, the number of transient errors grew slightly, and we encountered errors in two out of 10 instances that did not recover immediately.

At high severity, we found some diagnostic utilities functioned very slowly when a large number of multi-bit errors were inserted. For example, one utility scans the cache to count the number of data blocks that are dirty, and in so doing, accesses the cache many times. If those accesses touched a "broken" cell in memory, the code would try again, repeatedly, until a timeout occurred. This is because accessing global memory was done through a wrapper function, which checks for errors and optionally retries the access some number of times if there was an error. We found in the case of the utility in question, the optional "retry mode" was enabled. That caused the utility to run so slowly as to be practically unusable. Our recommendation was to invoke the wrapper such that it would only attempt accessing the cache once.

In another case, we found diagnosis utilities that swept the cache would terminate after encountering the first memory error. This slowed the recovery process faced by customer service engineers. We recommended modifying the utility to continue processing the entire cache. Overall we graded robustness of tools to be of medium quality.

Diagnosis at each level of severity was straightforward: replace the "faulty" memory component. We had no difficulties executing that solution at any of the severity levels, and therefore we graded simplicity and efficiency of the solution to be of high quality.

An important lesson we learned was that correlating faults generated in this experiment to a particular piece of the software could be very difficult. For example, if 10000 faults are generated, and one of them causes a cascading error that impacts some other functionality, the problem becomes a tedious search for which of the faults has an incorrect recovery path that produced the problem. An important enhancement to the fault injector software will be to improve logging of where the fault was generated. In summary, we found that our diagnostic tools had several areas of improvement in the area of robustness.

6. Discussion

We found that rehearsing the corrective action with a customer service engineer was a useful technique. During such "fire drills" we could observe firsthand whether the system's diagnosis utilities were effective and how soon it would take to locate and fix the problem. We are skeptical that automated scripts could be devised to simulate real-time debugging, except under greatly simplified conditions.

An organization's QA department may be the logical home for reliability "regression" tests. Such tests would have to be carefully developed, because untargeted fault injection can create bugs that are difficult to find and may waste developer time. Conversely, targeted tests may be too specific and numerous to be efficiently used by a QA group already burdened with tracking ever-changing software. In many cases it may be preferable for reliability testing to be done according to the responsible developer's sense of the software's sensitivity to a problem.

We note there is another class of behavior that may negatively effect system availability. An "upgrade event" is some user-initiated modification of the system, such as hot-swapping a drive or portion of memory. For a continuously available system, such operations should minimize availability degradation.

We consider these to be a separate class of problems for three reasons: (1) these operations are normally part of the manufacturers regression tests and significant losses of availability should be detected at that time; (2) maintenance upgrade events are rare (hot-swapping a major component of the system typically happens on the order of months or years); (3) Availability degredation is expected as some upgrade events necessarily have some effect (e.g. such as hot-swaping memory can lead to cache misses), and many utilities (e.g. gathering exhaustive drive or cache statistics) executed durring the maintenance procedure will necessarily sidetrack the system from completing I/O requests.

It would be desirable to quantify the entire system's reliability as a whole, rather than one component. Conceivably, this could allow two competing products to be compared against the same reliability benchmark. However, as noted by Prasad [10], this may not be possible. The range and variation of errors, and how they affect systems such as the one we test, is large. Separating a system such as ours into components (e.g. the cache) greatly simplifies the problem of reliability analysis [9].

We believe the problem of testing reliability of closed systems represents a fruitful area of research. It would be useful to obtain a sense of a system's reliability using some agreed upon criteria or standard, as is possible with performance tests.

Achieving this goal will be challenging. Measuring the reliability of a closed system (e.g. comparing competing products) is difficult [12]. Closed systems limit the scope of fault injection because software fault injectors cannot be written and hardware data-paths are unknown. Additionally, because the system's components (e.g. disk drives, memory cards), may be customized hardware, modifying them in a way to accurately generate a known fault is difficult. Finally, the management path (how system operators diagnose their systems in real-time) may be unobtainable.

Comparing the reliability of different storage systems is also complicated by the great variance of configuration options. Comparing "apples to apples" is very hard. To isolate testable similarities between competing machines, it is helpful to draw from experiences in performance testing. In such tests, certain subsystems play dominating roles, and other components are ignored. For example, in performance tests, the number and type of multiple input streams (SCSI, fast SCSI, fiber, ESCON, etc.), the number of drives, and the cache size are the most important variables that impact performance. The myriad of other potential configuration options provided (e.g. mirroring, RAID, dynamically attached disks), impact performance in a less obvious ways. We suggest evaluating two systems with the same configuration of dominant subsystems.

7. Related Research

Software fault injection (SFI) has frequently been used to generate untargeted faults. For example, by mutating code a programming error can be simulated [12]. Or, by simulating processor failure, a random event is created [2]. By repeatedly generating errors in this way statistics can be derived on reliability. Chen demonstrated that by using SFI, a positive development loop can be created to gradually improve the code [9].

The ISTORE project is studying reliability in storage systems, and has developed methodologies for availability benchmarking which we adapted for this paper. Their work has focused on examaning closed systems (e.g. Windows 2000) [1]. This work is directed torwards a development environment, in which any software fault instrument could be built for reliability measurement. Note that in development environment portability concerns (a frequent objection to SFI) are less of an issue.

Some work has been done to conceptually disassemble the complex software architecture that makes up a large storage system [6]. Kaaniche showed that once this is done, the problem of reliability measurement is somewhat simplified because lower "hierarchies" of software functionality may be viewed as a black box. Taking this useful perspective, our view in this paper was from the middle of the system, and the front and back end CPUs and devices were the "black boxes".

8. Conclusion

In this report, we have described our experiences in applying reliability benchmarks to the cache subsystem of a large disk array. We successfully found deficiencies in diagnostic utilities and located the fault severity levels at

which availability to the host became degraded. The work had a positive effect on the product as all of the deficiencies found were fed back to the designers who then made appropriate corrections. Two important lessons we learned are (1) testing techniques which focus both on particular algorithms and the overall system gave us a more complete picture of the system's reliability; and (2) maintainability is difficult to measure without human participation.

References

[1] Brown, A. Patterson D. "Towards Availability Benchmarks: A Case Study of Software RAID Systems." *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[2] Carreira, J. Silva, J. "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers." *IEEE Transactions on Software Engineering*. Vol 24, No 2. February 1998.

[3] EMC² Corporate Home Page: http://www.emc.com

[4] Kropp, N., Koopman, P. Siewiorek, D. "Automated Robstness Testing of Off-the-Shelf Software Components." *Fault Tolerant Computing Symposium*, pp. 230-239, June 23-25, 1998.

[5] Guedard, Y. Marneffe, L. Scheerens, F. Blanquart, H. Boyer, T. "Functional and Faulty Analysis: Some Experiments and Lessons Learned." *29th International Symposium on Fault-Tolerant Computing (FTCS-29)* Madison, Wisconsin, USA June 15-18, 1999.

[6] Kaaniche, M. Rmano, L. Kalbarczyk, Z. Iuer, R. Karcich, R. "A Hierarchial Appraoch for Dependability Analysis of a Commerical Cache-Based RAID Storage Architecture." *The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*. 3 - 25 June, 1998.

[7] Koopman, P. "Toward a Scalable Method for Quantifying Aspects of Fault Tolerance, Software Assurance, and Computer Security." *Post Proceedings of the Computer Security, Dependability, and Assurance (CSDA'98)*, 11-13 Nobember 1998.

[8] Michael, C. "On the Uniformity of Error Propagation in Software". *Proceedings of the 12th Annual Conference on Computer Assurance (COMPASS '97)*. Gaithersburg, MD. 1997.

[9] Ng, W. Chen, P. "The Systematic Improvement of Fault Tolerance in the Rio File Cache." *Proceedings of the 1999 Symposium on Fault-Tolerant Computing (FTCS)* , June 1999.

[10] Prasad, D. McDermid, J. "Dependability Evaluation using a Multi-Criteria Decision Analysis Procedure." *Proceedings of the Dependable Computing for Critical Applications*. January, 1999.

[11] Talagala, N. Patterson, D. "An Analysis of Error Behavior in a Large Storage System." *The 1999 IPPS Workshop on Fault Tolerance in Parallel and Distributed Systems*.

[12] Voas, Jeffrey. McGraw, Gary. "Software Fault Injection". Wiley Computer Publishing, New York, 1998.

HP Scalable Computing Architecture

Randy Wright
Arun Kumar
HP Platform Software Architecture Lab

Abstract

The HP V-Class server family provides up to 32 processors and 32 GB of memory in a single cabinet. Scalable Computing Architecture technology allows multiple V-Class cabinets to be interconnected, forming a single cache coherent non-uniform memory architecture (ccNUMA) system providing up to 128 CPUs and 128 GB of memory. This paper discusses some interesting aspects of the Scalable Computing Architecture and the changes made to the HP-UX kernel to operate in this environment.

1. Introduction

The Scalable Computing Architecture (SCA) was proposed to extend the scalability of Hewlett-Packard's high end V-Class machine. The V-Class is a crossbar based symmetric multiprocessor (SMP) machine. The SCA machine connects up to four V-Class nodes using a high-speed high bandwidth interconnect to conform to the ccNUMA architecture.

Normally, Hewlett-Packard's UNIX machines conform to a well-defined architecture as specified in [1] and [2]. The V-Class machine [3, 4] did not fully conform to this architecture. However, by emulating the PA-RISC architecture in the firmware layer, the HP-UX operating system could be supported on the V-Class. The architectural anomalies of the V-Class platform created unique challenges in supporting HP-UX on the SCA platform. This paper describes the modifications made to the HP-UX kernel to address some of these anomalies in the hardware and to extend the capabilities of the V-Class to meet the requirements of the ccNUMA architecture.

At the time we began the SCA operating system work, the V-Class hardware platform was already running HP-UX in a single node configuration. In addition, the SPP-UX operating system, based on the Mach [15] micro-kernel, was running on essentially the same platform (referred to as X-Class) in a multi-node configuration. Leveraging the technology used on the X-Class architecture to expand HP-UX server product line to 128 processors was viewed at that time as a cost effective way to expand HP's UNIX server line.

2. V-Class Hardware Architecture

Figure 1 is a high-level block diagram of a single cabinet V-Class HP computer system.

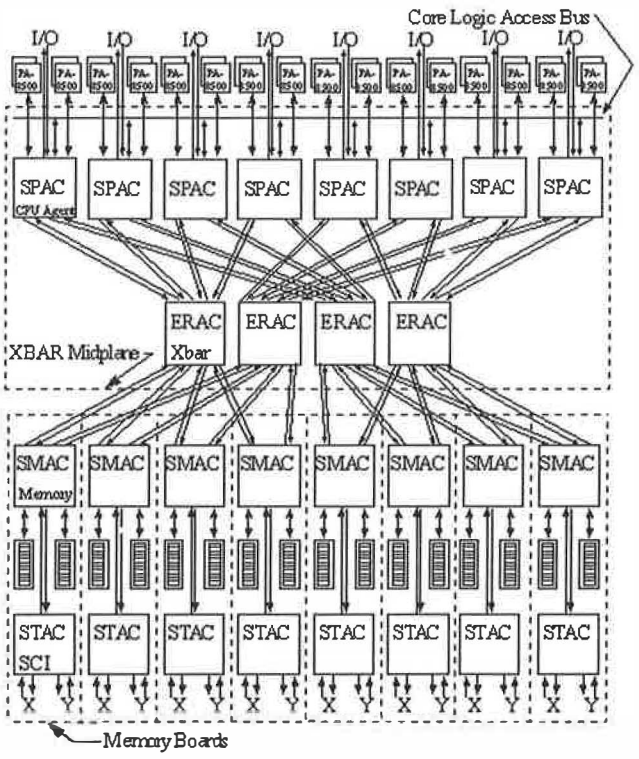


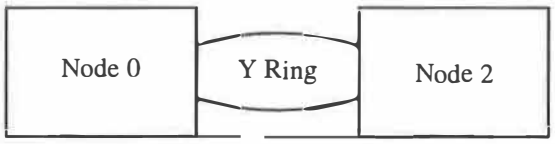
Figure 1: V-2500 Node

The V-Class system is a crossbar-based machine. On the V-2500, four processors are attached via dedicated busses (termed Runway busses) to a processor agent chip (SPAC). From each SPAC there is attached a PCI

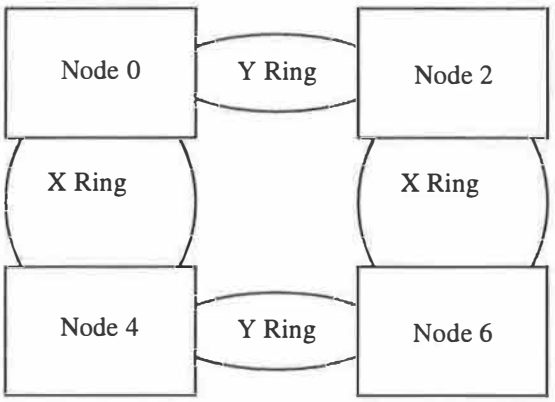
interface chip (SAGA) and bus. Memory is interleaved across banks that are managed via memory access controllers (SMAC). A single node V-2500 system is compliant to the PA-RISC platform architecture, implementing a coherent I/O SMP platform. A maximum configuration V-2500 consists of 32 CPUs, 8 SPACs, 8 PCI busses (with 3 slots each), and 8 memory boards (SMACs) providing 4 to 32 GB of memory. Not shown in the diagram is the utilities board. It contains resources used in bootstrapping, reset, configuration, and diagnostic functions. This board contains an RS-232C port used to connect to the system console, an Ethernet connection used for manufacturing and system diagnostic purposes, an LCD display, non-volatile and flash RAM used to hold system firmware and configuration settings, and static RAM used by the system firmware. Each SPAC has access to the utilities board of its containing node.

At the bottom of Figure 1, we see that a Toroidal Access Controller chip (STAC) is attached to each SMAC. This chip implements a variation of the Scalable Coherent Interconnect (SCI) protocol over one to two rings. The STACs and rings are sometimes referred to as the Coherent Toroidal Interconnect (CTI). This interconnect allows V-Class systems to be scaled to multiple nodes (where each node is a cabinet of up to 32 CPUs). Most processor or I/O transactions are conducted through a SPAC, through the crossbar and to a SMAC. If the transaction refers to a component not on the local node, the SMAC forwards the request to its corresponding STAC that in turn traverses the correct ring to a remote node STAC. On the remote node, the STAC sends the transaction to its associated SMAC where it is resolved or forwarded through the remote crossbar port to the appropriate component. Inter-node requests are interleaved across multiple SCI rings in the appropriate ring set (each STAC resides on two rings labeled X and Y). Inter-node memory coherency is managed on a 32-byte line basis. Sample V-Class SCA system configurations are shown in Figure 2. The node numbering contains hardware topological and routing information; this is the reason that nodes in a 4-node system are numbered 0, 2, 4, and 6.

The multi-node V-Class architecture includes a second level inter-node cache, referred to as the CTI cache or the network cache. This is a region of local node memory that is interleaved across memory controllers and is used to cache remote node accesses. The processors cannot access memory used in the CTI cache. Each node typically has 16M to 512M of its local memory used in the CTI network cache. The CTI cache size is



Two Node Configuration



Four Node Configuration

Figure 2: Sample SCA Configurations

configured at boot and is not dynamically configurable; a reset is required to change it.

The precise layout of coherent memory space depends upon the number of memory boards present, size of the DIMMs used on the boards and the amount of memory configured as CTI cache. However, as the node number is encoded into the high order bits of the physical memory address, the memory space of node 0 starts at 0, node 2 starts at 64G, node 4 at 128G, and node 6 at 192G.

All of coherent memory is globally accessible except the memory that is referred to as node private or force node id memory. When the V-Class system was architected, it was anticipated that each node would need some local memory below 4G for firmware and specialized software. To address this need, a special memory access mode known as force node id was introduced. This feature causes the first 128M of each node to be locally mapped, providing firmware with desired local 32 bit-accessible memory on all nodes but resulted in a peculiar physical aliasing property. Consequently, these regions of memory are only accessible by CPUs on the local node.

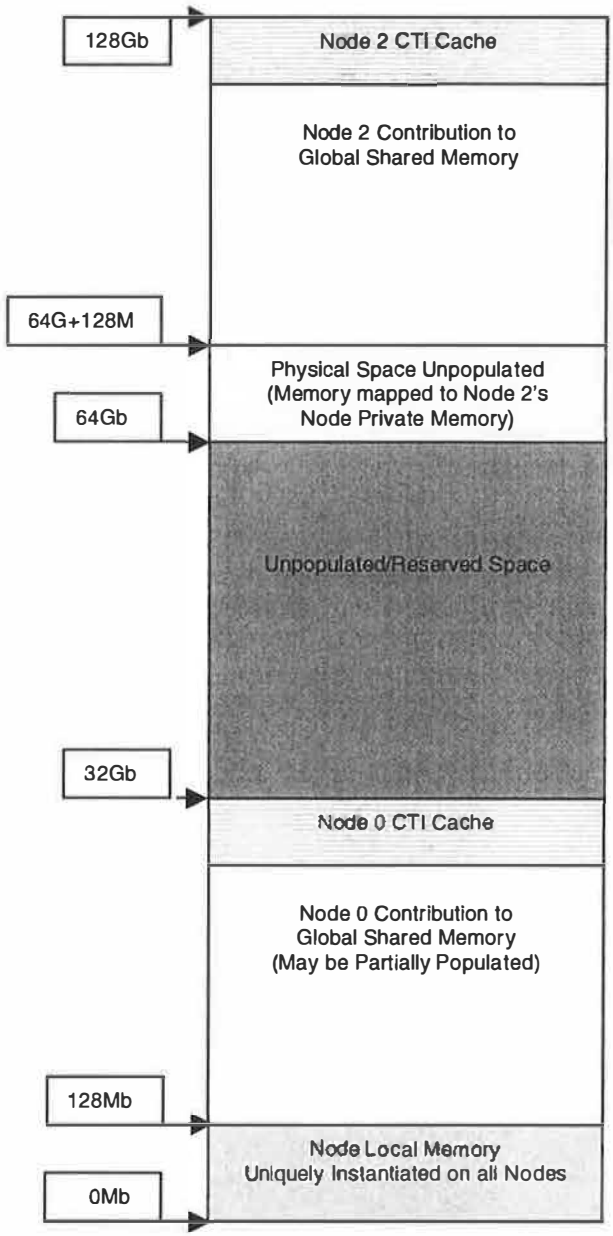


Figure 3: SCA Physical Memory Space

Figure 3 shows the SCA coherent memory address space.

3. SCA Software Architecture

The software development work to support HP-UX on the SCA platform was determined by asking the simple question, "How is a Multi-Node V-Class complex not like a symmetric multi-processor (SMP) system?" and addressing the anomalies so identified. The following discussion summarizes these anomalies and the steps taken to address them.

3.1. Device Management and I/O

One software design goal is to maintain kernel independence from the details of the underlying NUMA system. However, for scalability reasons, it is sometimes necessary to be able to know about the NUMA aspects of the system. To reconcile these conflicting goals, we used an abstraction termed a *locality domain* to provide a rough gauge of "local" versus "remote." A locality domain is a collection of resources including processors, memory, and possibly I/O devices. Equal latency to memory is the characteristic relating resources grouped into a locality domain. On a V-Class SCA system, locality domain boundaries always correspond exactly to the boundaries of a physical node, but the abstraction allows future architectures to define these boundaries more flexibly.

So that this locality domain abstraction could have long-term applicability to other hardware products, it is desirable to prevent exposing hardware-independent code to hardware specific concepts such as the non-sequential node numbers of the V-2500. Therefore, a logical mapping was instituted to insulate high-level software from the hardware-dependent node numbering of V-Class.

The design resulting from these considerations compartmentalizes the hardware node numbering in a low-level hardware-dependent layer, exporting a sequential locality domain or "logical node" numbering to higher-level software when such knowledge is required. This logical node numbering results in a four-node system with logical nodes 0, 1, 2, and 3.

In the following discussion of I/O we will discuss two different methods of identifying I/O devices: Hard Physical Address (HPA) and hardware path. A device's HPA is a unique 64-bit address used internally by system software to identify a device and (in most cases) to actually map it into the processors' address space. We will also discuss hardware paths, which identify devices by their relative location in the system hierarchy of adapters, controllers, and device instances. The hardware path identifies each element in the system with a hardware address that is unique with re-

spect to its superior hardware element; it is not constrained to fit uniquely in a 64-bit address.

Software normally performs I/O on PA-RISC systems by manipulating Control and Status Registers (CSRs), which map into the processors' address space. On a Multi-Node SCA complex, many of the CSRs are not globally accessible across nodes. PCI I/O controller CSRs are accessible only by CPUs on the node containing that PCI controller. Utilities board functions are accessible only from the node containing that utilities board. Most other non-CPU system Core Electronics Complex (CEC) registers are globally accessible. The CPU and most system gate arrays may be addressed globally with unique addresses, as the node number is encoded within the physical address. However, PCI CSRs are mapped node private only; they are not unique complex-wide. For example, all PCI card CSRs for slot 0 of the first PCI bus of each node will have identical physical addresses.

For I/O devices accessed through HP-UX drivers dealing with this aspect of the architecture requires driver software to use process management primitives to migrate its execution to a CPU on the node containing the device to be accessed.

For block devices as accessed via the file system, the solution for binding to the correct node took advantage of existing code that preferred (for cache efficiency reasons) to run the base level code of a device driver on the processor that is designated to handle interrupts for that device. The preference, previously implemented as optional by this I/O forwarding code, simply was made mandatory for the SCA system.

Two sub-cases must be considered to provide direct access to block and character devices, as the kernel supports both legacy uniprocessor drivers as well as multiprocessor-safe drivers. For multiprocessor safe drivers, the I/O system was modified to assure that the interrupt delivery is targeted to a processor on the correct node (that is, the node containing the device) when initializing the device. Base level processing for a multiprocessor-safe driver is bound to a processor on the correct node at the time a request is made; this binding lasts only for the duration of a specific request but has the tendency (unless the requesting process has otherwise specified a processor binding) to migrate processes to the node containing I/O devices they use most heavily.

For uniprocessor drivers, HP-UX provides an emulation wrapper that binds all such drivers to the first processor booted (termed the *monarch* processor). By so doing, this wrapper technique forces traditional uniprocessor serialization when executing base level requests, and by forcing the interrupt handlers to run on the monarch processor as well, provides a very simple paradigm to protect against reentrancy. For the SCA architecture, however, this model had to be improved or uniprocessor driver support could not be provided.

The existing uniprocessor binding mechanism could not simply select the monarch as the emulated uniprocessor target on an SCA system due to the restriction that I/O CSR access is usually not possible across nodes. We attempted to create a generic mechanism that by using a semaphore per device manipulated in a generic wrapper could provide the serialized semantic required by uniprocessor drivers. After providing the serialization required, we could use the process management primitives to bind the executing thread to the appropriate processor. This seems at first to be a simple task, but when a driver sleeps (for instance, awaiting input from a serial port), the generic semaphore is still held by the driver and as a result, no other process can enter the device driver. A semaphore type that is automatically released when the holder sleeps can solve this problem, but an efficient implementation of such a semaphore type could not be provided in the time available. As a result, support for uniprocessor device drivers could not be maintained for SCA systems.

There are restrictions on access to the utilities board functionality that are closely related to those described on CSR accesses. The utilities board resides in non-coherent system CSR space and is accessible only by CPUs on the local node. Therefore, CPUs on one node cannot access the console, NVRAM, or firmware storage areas on another node.

The NVRAM and firmware storage restrictions do not greatly impact the kernel. The kernel retrieves data from firmware only through architected firmware interfaces. The SCA system design calls for all significant firmware configuration data to be stored on node 0 only. By using appropriate scheduling primitives, software assures that the firmware interfaces are invoked from node 0 only, and so this restriction is made a non-issue.

The utilities board also contains the serial port used as the console. Just as is done for other HP-UX device drivers, the console driver uses scheduling primitives to bind to a CPU on the correct node, which for the HP-UX system console is always node 0. (There are

serial ports on each node's utilities board, but those on nodes other than 0 are normally unused).

A reliable global reset is required to reboot the system; without a reliable reset, one or more processor may not participate correctly in the next system boot. The V-Class hardware reset mechanism is implemented by writing to a specific CSR on the system utilities board, and it resets only the local node. But hardware does not provide a system-wide atomic reset on a Multi-Node SCA system. No HP-UX kernel change was required to address this anomaly. Instead, the V-Class Processor Dependent Code (PDC) firmware was modified so that a firmware-initiated reset request would reset the entire complex, rather than a single node. The firmware accomplished this by using a diagnostic communication interface between nodes of which the kernel is totally unaware.

The V-Class hardware implements two distinct methods of addressing non-I/O device CSRs, referred to as node-local and global modes. When HP-UX was initially ported to the single-node V-Class platform, a decision was made to use the simple node-local addressing mode for processor HPA (Hard Physical Address) access, as well as other ASIC CSR accesses. This addressing format is useful for accessing CSRs on the current node of execution, but cannot access CSRs on remote nodes.

Inter-node access requires a more general global address format for CSRs. This uses a wide-mode (64-bit) address format that the underlying single-node firmware does not supply.

In addition to correct CSR hardware addressing, each module must be uniquely identified by its HPA. The HPA returned for a memory module is a Coherent Memory address. The HPA returned by firmware for CPU modules during device discovery is the physical address of the CPU's CSR block. These CSRs are located in the non-I/O CSR space. The path through the crossbars is included in the address. Since the kernel does not know how to route an address to a remote node through the crossbars, it relies on the firmware to provide it with an HPA that will work for the platform configuration. Because the path can be different from one node to another, it is required that all nodes of a Multi-Node V-Class platform are fully populated with SPACs, SMACs, and STACs for this to work correctly.

The address space used to access PCI I/O cards and core device I/O CSRs is termed Local I/O Space (core devices are those built into the system, not attached via a peripheral card). Access to this space is available only to CPUs on the same node as the PCI bus adapter or core I/O device. The method used to generate addresses for the local I/O space does not permit the node number to be embedded into a 64-bit or 40-bit physical address. However, the HPA returned by PDC firmware during device discovery for these modules is not a valid CSR address. Therefore, the HPA is used only to identify the module and not to access CSRs. This allows us to treat the address as if it were a coherent memory address for the purposes of generating a unique 64-bit HPA.

To support V-Class SCA I/O, a hardware path must uniquely identify each I/O device. The hardware path is generated from the device path supplied by PDC firmware in response to a PDC_SYSTEM_MAP command during module discovery. On a single-node system, the device path returned by PDC does not contain an indication of the node number containing the module. It is possible - in fact, quite likely - that when two or more nodes are combined to form an SCA system, two or more devices will have the same path, making it impossible to distinguish between them. Therefore, an indication of the physical number of the node containing the I/O module has been encoded in the first component of each device in an SCA system.

For example, the hardware paths for single node V-Class devices have the I/O adapter number - the component called the SAGA on V-2500 systems - as the first component of the hardware path. There may be up to 8 adapters per node, numbered from 0 to 7. Each SAGA controls up to 3 PCI slots, numbered 0 to 2. For instance, 2/1/0.4.0 would refer to a SCSI disk attached to SAGA 2, PCI slot 1, SCSI id 4. If this same SAGA/slot/SCSI-id occurs on node 2, adding the node number times 32 to the I/O adapter number uniquely identifies that adapter within the complex. Extending the previous example, 66/1/0.4.0 would represent SAGA 2 of node 1, slot 1, and SCSI target 4.

The HP-UX kernel typically derives system timing information from an internal control register that implements an interval timer (termed the *itmr*). Each processor contains such an itmr, and in almost all HP PA-RISC systems, these itmrs are synchronized. On an SCA V-Class system, the interval timers (itmrs) within a node are all generated from a single crystal clock source and so are synchronized within that single node. However, the clocks are separate for each node. This means that the itmrs cannot be used to generate synchronized timing information across nodes of the com-

plex, as they can drift with respect to one another over time.

Some code needing timing of events precisely across the system that previously used the processor itmr registers was modified to use a V-Class specific feature. The V-Class hardware provides a 1 microsecond period counter known as the Time of Century counter. This counter is 64 bits wide and globally synchronized in hardware. The maximum inter-node skew is 1 microsecond. This provides system software with a globally consistent view of time without requiring synchronized processor clocks across nodes.

For generic time accounting purposes, we did not want to introduce V-Class specific code throughout the kernel. To support a few other systems with asynchronous clocks, the kernel already had the ability to compensate for itmr drift by maintaining a synchronized per-processor bias tracking the drift of each processor's itmr against the master (or monarch) processor. We were able to take advantage of this existing mechanism and extend the synchronization code to use the Time of Century hardware as a global reference to precisely compensate for the clock drift between nodes.

The PDC firmware of each node had no visibility of the other nodes' resources; each node's PDC initially operated totally independently. It was immediately obvious when starting the SCA kernel design that compensating for this major "non-SMP-ness" explicitly in the kernel would become very intrusive if handled with ad hoc methods. Consequently, a "front-end" or facade layer was designed to produce PDC firmware functionality supporting multi-node SCA in a transparent manner. This component is called Multi-Node PDC layer. It is linked into the HP-UX kernel and mapped specially into the shared memory region. The Multi-Node PDC Front-End provides the following functionality:

a. A unified module table, with some limited exceptions: only node 0 core I/O modules are included; as a result the kernel has no access to console or diagnostic LAN ports of non-zero nodes, and non-zero node memory modules do not include node private memory.

b. The node local CSR address format used by underlying PDC firmware is modified to use the complex-wide global addressing format containing an explicit node number

c. Device paths encoded to include the module's node number.

d. Runtime support for PDC requests that require a remote call to another node.

e. Ability to perform inter-node memory copies from one node's local memory to another node's local memory for specialized initialization and crash dump requirements.

The resulting layer of software should be considered logically as a part of the system's firmware. Whereas it is linked into the kernel image, after an initialization call, it is never directly called again. Instead, it replaces the contents of the location used to vector into the PDC firmware, directing firmware requests to its own entry and then chaining, where appropriate, to the underlying PDC firmware. This design allows the Multi-Node PDC layer to be updated when the kernel is updated. This design choice was made to simplify both development of the firmware layer and field upgrades of existing single-node systems to SCA configuration in the field, and its design succeeded in so doing. However, it requires a logical separation of responsibility so that the Multi-Node PDC layer cannot make calls back into the kernel; if this is violated (for instance, by inadvertent use of a compiler-generated runtime library support function call), the firmware layer may fail during system boot of non-zero nodes. Maintaining this separation has proven to be difficult over time, and since the consequences of violating the required layering are both profound and difficult to diagnose, the design choice is questionable in hindsight as a permanent part of the kernel.

3.2. System Characteristics

One very significant way in which a V-Class SCA system differs from a traditional SMP system is that there exists a special memory region on each node termed *force node-id* or *node private* memory (as discussed above in Section 2) This memory consists of the first 128M of each node, and is accessible only by CPUs on the containing node. The node private memory is unique in its combination of attributes:

a. Spatial locality: In a NUMA sense, access is guaranteed to be efficient because this memory is by definition local to the accessing node.

b. Limited accessibility: only processors on the node may access this memory. This is enforced by the hardware.

c. Uniquely instantiated per node: this refers to the fact that a given address of memory may contain different values on different nodes. This can be very useful for variables that are location dependent; an obvious example is a single location containing the node number for each node. It allows the use of a single variable in the code to be used differently depending upon the node context of the accessing CPU.

The kernel text segment is replicated, one copy per node, in an effort to reduce instruction fetches between nodes. This lowers latency and at the same time does not pollute the CTI caches with kernel text. The node private memory of each node is used to contain that node's copy of the kernel text. (The obvious but much less efficient alternative to this replication would have required loading the kernel at or above the 128Mb boundary, in globally shared memory.)

To accomplish the replication, a new function *rm_ktext_replicate*() is called during system boot. If running on a platform other than Multi-Node V-Class, *rm_ktext_replicate*() performs no action and returns immediately. On Multi-Node V-Class, the function copies the kernel text segment from the node private memory area on node zero to the node private memory area of all other nodes in the system.

The actual replication of kernel text is done near the very end of the initial phase of the boot sequence. The replication is done after the point at which the kernel has performed all platform dependent optimization decisions. In addition to copying the kernel text itself, page zero (which contains data shared between the kernel and PDC firmware) also is replicated to all other nodes. The memory copy operation is done via a firmware interface added specifically for V-Class SCA systems.

The layout of text and data within the SCA kernel is different than that of a traditional HP-UX kernel. As part of exploiting the node private memory architecture to replicate the kernel text, the SCA kernel is linked specially, specifying an alternate memory map to the linker. A mapfile specifies that the text segment remains at a low address in physical memory, and so is in node private memory, but the data segment is placed just above 128M, and so is in global shared memory.

The HP PA-RISC processor performs translations from virtual to absolute addresses using a hardware structure called the Translation Lookaside Buffer (TLB). The purge instruction TLB (*pitlb*) and purge data TLB (*pdtlb*) instructions are used to manage processor TLB entries. On the SCA V-Class platform, these transactions are not broadcast between nodes. Another limitation is that there can only be one outstanding purge per node.

To address these architectural limitations, the kernel implemented an inter-node RPC (Remote Procedure Call) interface that is initiated by a software interrupt when a TLB modification is performed. The interrupt is serviced by dedicated kernel handler code to perform the needed purge operation on each node. Challenges addressed in the design include the need to make the service code very efficient and responsive, yet minimize intrusion of the software on architecturally conforming HP systems.

Processor caches in all multiprocessor systems must be managed with care. For PA-RISC based systems, the flush instruction, flush data, and purge data cache instructions (*fic*, *fdc*, and *pdc*) are used to maintain consistent cache state among CPUs. In the SCA V-Class platform, these flush instructions are not broadcast between nodes. The instruction cache is a particular problem, as there does exist a method to cause a global data cache flush of a line of memory. There is a special host op instruction supported by the PA-8x00 processors that is passed nearly transparently to the system gate arrays. This instruction will globally flush a line of memory from all data and network caches. It is referred to as the NCFG instruction (Network Cache Flush Global). The host op instruction is not precisely interchangeable with the existing architected flush instructions, so special software handling was required.

The instruction cache anomaly required handling user space pages via what we called *write-execute demotion*; a page of user accessible memory could not simultaneously be granted both write and execute permission. Such pages are internally recorded in the virtual memory system as allowing write and execute, but set up in the hardware as only write or execute. The permissions are switched from one to the other in the kernel's trap handler upon an attempt to write an executable page or to execute a writable page. There was concern that this mechanism might be too inefficient for applications that engage in lots of runtime patching. This has not been found to be a problem based on our limited experience, as most such instruction patching or generation occurs only once during a process's life-

time. However, the implementation does produce an inconsistency that the application can detect via the *probe* instruction, which sees the "real" hardware permission – only write or execute - rather than the emulated write and execute permission. Consequently, we created library function interface for use by any application that needed to get the correct result on all architectures. This interface is written so as to query the virtual memory system for the permissions on the SCA V-Class system, rather than trusting the *probe* instruction.

Kernel pages can be handled more simply, as the kernel functions used to flush caches and purge the TLB could be modified and/or patched as required. The modifications to use the RPC mechanism alluded to above were in fact implemented using a single paradigm, sharing data structure definitions and software strategies. When a cross node purge or flush operation is required, the processor generates a pre-designated interrupt type (which differs for purge and flush) to a pre-designated partner processor. The data structure involved records the address range and operation to be affected. The interrupt is serviced at a high priority level; the requestor, in order to maintain the required architectural semantics of the flush or purge operation, must busy wait until the partner completes the requested operation.

There are rare - but quite possible – cases wherein matched pairs of processors on two different notes, each the pre-designated RPC partner of the other, could encounter a deadlock. If one processor holds a spinlock while requesting a flush operation (as sometimes occurs in the page directory manipulation) and its RPC partner processor is in fact waiting for the same spinlock, the result would be a deadlock, since the waiting processor has disabled interrupts in the spinlock acquisition code. To avoid this potential deadlock, we modified the kernel's spinlock routines so that threads waiting for spinlocks periodically check for outstanding remote TLB purge or cache flush requests and service any so detected, thereby avoiding deadlock.

Finally, the virtual memory system was modified to reflect the ccNUMA memory system attributes. Roughly stated, the latency to lines of memory within a node is 500 nanoseconds. This can blossom 2 to 4 times for remote accesses (2.2 microseconds). In addition, access time is not strictly bounded; it can be dependent upon SCI ring contention. The actual coherency state of a line (home, dirty, read shared) and the operation being performed (read, write, flush) on the line also impact the latency.

As more NUMA architectures are anticipated in the future, the virtual memory system was enhanced with knowledge of the locality domain abstraction to allocate memory as efficiently as possible. A first-fault strategy was chosen to instantiate pages for a process from the memory pool of the first locality domain (node in this case) to generate a fault on the process's address space.

The HP-UX process management system was enhanced to efficiently employ the NUMA characteristics of the V-Class SCA architecture. The scheduler was given knowledge of the system's locality domains so that it could effectively place newly created threads as well as appropriately avoid migrating threads across locality domains.

The primitives available for locating threads and processes on specific locality domains and CPUs within the system were enhanced. Particular attention was paid to needs by the I/O system, which may need to migrate threads onto processors appropriate to the node containing a particular physical device.

A new utility *mpsched* was provided to explicitly control thread and process placement within the complex. The policies provided are

a. Round robin launch policy, in which launches of child threads or processes alternate among all localities until all localities have been selected once, then starts over as needed.

b. Least loaded launch policy, under which child threads or processes are launched on the least loaded node in the system at the time of creation.

c. Fill first launch policy, wherein successive processes or threads are launched on the same locality domain as their parent until one has been launched on each processor in the locality domain; at that point, new threads are created on the next locality domain.

d. Packed launch, under which successive threads or processes are launched on the same locality domain as their parent; a different locality domain is never selected.

The default scheduling decisions, at a coarse level, attempt to start new processes on the least loaded node and new threads of an existing process on the node containing the creating thread. In addition, the load is periodically balanced among nodes when the imbalance exceeds some threshold or some node becomes idle.

4. Related Work

One finds a consensus in published literature that approaches beyond traditional SMP are necessary to achieve cost effective scaling of parallel computer systems. The approaches to implement such scaling alternatives vary greatly. Descriptions of hardware implementations in the literature include the directory-based ccNUMA hardware designs of DASH [5] and Alewife [6] as well as Cache Only Memory Architecture (or COMA), exemplified by Kendall Square's KSR-1 and analyzed in [7] and [8]. A hybrid of ccNUMA and COMA, termed Reactive NUMA, is described in [9]. An approach using a Cache Coherent Network of Workstations (ccNOW) is described in [10]. Yet another approach is to omit caches altogether, compensating for the resultant latency by instruction-level parallelism, exemplified by TERA [16]. Studies of memory and process placement in NUMA systems have been extensively described and analyzed in [11], [12], [13], and [14].

Operating systems for NUMA architectures may be monolithic SMP systems modified for NUMA such as our HP-UX port, or micro-kernel based. Micro-kernel based solutions have included Mach [15]. The SPP-UX Mach-based operating system from the HP Convex Division used a message-passing paradigm to communicate between distinct micro-kernel instances running on each node. Other proprietary operating systems are described in many of the above hardware references.

5. Conclusions

In this paper we described the issues encountered and the solutions used for the SCA V-Class project. Some of the areas addressed are platform-specific and unique to the V-Class architecture. However, other areas related to NUMA memory management techniques and process and thread management in a NUMA environment can be leveraged on future HP platforms.

Internally, the V-Class SCA platform is being used to study and improve the performance and the scalability of the HP-UX Operating System. We expect that our experience and learning from the V-Class SCA platform will enable the HP-UX operating system to scale to even larger configurations in future.

6. References

[1] *PA-RISC 2.0 Architecture*, Jerry Kane, ISBN 0-13-182734-0, Prentice Hall; also available online at http://devresource.hp.com/devresource/Docs/Refs/PA2_0/index.html

[2] PA-RISC 2.0 Firmware Architecture, http://devresource.hp.com/devresource/Docs/DocLibrary.html

[3] HP V-2500 Reference Guide, online at http://docs.hp.com/HP-UX/systems

[4] The Evolution of the HP/Convex Exemplar; Brewer, T., Astfalk, G. Convex Div., Hewlett-Packard Co., Richardson, TX, USA, *Proceedings of Compcon 1997*.

[5] The DASH Prototype: Implementation and Performance; Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta and John Hennessy; *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, Pages 92 – 103.

[6] The MIT Alewife Machine: Architecture and Performance; Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Kenneth Mackenzie and Donald Yeung; *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, Pages 2 – 13.

[7] Comparative Performance Evaluation of Cache-coherent NUMA and COMA Architectures; Per Stenstrom, Truman Joe and Anoop Gupta; *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, Pages 80-91.

[8] An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessors; J. P. Singh, T. Joe, J. L. Hennessy and A. Gupta; *Proceedings of the Conference on Supercomputing '93*, 1993, Pages 214-225.

[9] Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA; Babak Falsafi and David A. Wood; *Proceedings of the 24th International Symposium on Computer Architecture*, 1997, Pages 229-240.

[10] The S3.mp Architecture: A Local Area Multiprocessor; A. Nowatzyk, M. Monger, M. Parkin, E. Kelly, M. Browne, G. Aybay and D. Lee; *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1993, Pages 140-141.

[11] NUMA Policies and Their Relation to Memory Architecture; William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler and Alan L. Cox; *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991, Pages 212-221.

[12] Exploiting Operating System Support for Dynamic Page Placement on a NUMA Shared Memory Multiprocessor; Richard P. LaRowe, James T. Wilkes and Carla S. Ellis; *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 1991, Pages 122-132.

[13] An Analysis of Dynamic Page Placement on a NUMA Multiprocessor; Richard P. LaRowe, Mark A. Holliday and Carla Schlatter Ellis; *Proceedings of the 1992 ACM SIGMETRICS and PERFORMANCE '92 International Conference on Measurement and Modeling of Computer Systems*, 1992, Pages 23-34.

[14] Dynamic and Static Load Scheduling Performance on a NUMA Shared Memory Multiprocessor; Xiaodong Zhang; *Proceedings of the 1991 International Conference on Supercomputing*, 1991, Pages 128- 135.

[15] An extensive archive of published and unpublished papers on Mach is found at: http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mach/public/www/mach.html

[16] Exploiting Heterogeneous Parallelism on a Multithreaded Multiprocessor; Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield and Burton Smith; *Proceedings of the 1992 International Conference on Supercomputing*, 1992, Pages 188-197.

Stub-Code Performance is Becoming Important

Andreas Haeberlen Jochen Liedtke Yoonho Park Lars Reuther Volkmar Uhlig

University of Karlsruhe
System Architecture Group
76128 Karlsruhe, Germany
(haeberlen,liedtke,uhlig)@ira.uka.de

IBM T.J.Watson
Hawthorne, NY 10532
yoonho@us.ibm.com

Dresden University of Technology
Department of Computer Science
01062 Dresden, Germany
reuther@os.inf.tu-dresden.de

Abstract

As IPC mechanisms become faster, stub-code efficiency becomes a performance issue for local client/server RPCs and inter-component communication. Inefficient and unnecessary complex marshalling code can almost double communication costs. We have developed an experimental new IDL compiler that produces near-optimal stub code for gcc *and the L4 microkernel. The current experimental* IDL^4 *compiler cooperates with the* gcc *compiler and its x86 code generator. Other compilers or target machines would require different optimizations. In most cases, the generated stub code is approximately 3 times faster (and shorter) than the code generated by a commonly used portable IDL compiler. Benchmarks have shown that efficient stubs can increase application performance by more than 10 percent. The results are applied within IBM's* SawMill *project that aims at technology for constructing multi-server operating systems.*

1 Motivation

Multi-server and component-based systems are promising architectural approaches for handling the ever-increasing complexity of operating and application systems. Components or servers (and clients) communicate with each other through cross-domain method invocations. Such interface method invocations, if crossing protection boundaries, are typically implemented through the inter-process communication (IPC) mechanisms offered by a microkernel.

Firstly, component interaction in such systems has to be highly efficient. Therefore, for over a decade, performance-oriented research focused on microkernel construction, in particular IPC performance, finally resulting in acceptable IPC overheads (100...200 cycles)[6, 2].

Secondly, component interaction has to be conveniently usable from an application programmer's perspective. This requirement led to the development of interface-definition languages (IDLs), e.g. Corba IDL [7], DCOM [3] and their corresponding IDL compilers. From interface procedure/method definitions, such compilers generate stub code that marshals parameters on the client side, communicates through IPC or RPC kernel primitives with the server, unmarshals the parameters on the server side, invokes the corresponding server procedure/method, etc. As a result, a programmer can specify and use remote interfaces as easily as internal ones.

So far, IDL-compiler research has focused more on generating code in a portable and adaptable way than on producing efficient stubs. In fact, stub-code performance was insignificant for early microkernels that required multiple thousands of cycles per IPC. However, with high-performance IPC, stub-code efficiency becomes an issue.

For example, when using the Flick IDL compiler [4] for the *SawMill* Linux file system [5], we found that the generated user-level stub code consumed about 260 instructions per read request. When reading a 4K block from the file system, the stub code adds an overhead of about 17% due to stub instructions. (The stub code may also generates further indirect costs through side effects such as cache pollution.) For an industrial system, such overheads can no longer be ignored.

Hand coding of the aforementioned stub resulted in 80 instead of 260 instructions. Although this was a singular experiment, it gave us some evidence that improving stub-code generation might be worthwhile. The potentially achievable reductions justified a compiler-construction experiment to explore whether near-optimal stub code can be generated at reasonable costs.

This paper describes the resulting IDL^4 compiler that generates code for *gcc* on x86 and the L4 microkernel. The current IDL^4 is a prototype that purely focuses on generating efficient code. Portability and adaptability

are ignored and remain a topic for future work.

Structure of the paper

This paper reports on progress that has been made with IDL^4, an experimental IDL compiler for the L4 microkernel. Section 2 sketches prerequisites for understanding the subsequent discussion such as IDL syntax, L4-IPC mechanisms, and our experiences using the Flick IDL compiler in the *SawMill* project. Section 3 describes the stub-code model that was designed for the IDL^4 compiler, and Section 4 illustrates the code-generation principles. Finally, Section 5 reports on the achieved stub-code quality, Section 6 discusses the costs of adapting the system to other processor architectures and compilers, and Section 7 concludes.

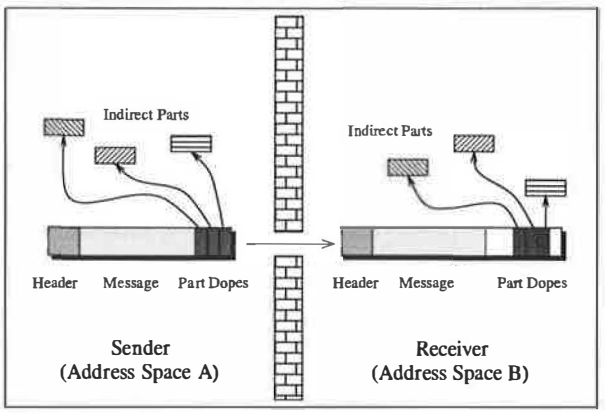


Figure 1. Complex memory message including indirect strings.

2 Prerequisites

2.1 L4/x86 IPC

L4's [6] basic communication paradigm is synchronous IPC. Typical operations are *send, receive, call* (atomic send&receive), and atomic *reply&wait.* Rich message types help to improve end-to-end IPC performance:

Register messages consist of a small number of 32-bit words that are sent and received in general purpose registers. On the x86 platform, up to 3 words (plus sender id and message descriptor) can be transferred as a register message. As there is no need for copy operations across address space boundaries, register messages have the lowest IPC costs, e.g. 180 cycles on a Pentium III 450 MHz.

Memory messages can be used to copy longer messages from the sender's address space to the receiver. Message size can be up to 2MB; however, this mechanism is slower than a register message because it involves copying from/to memory, and the kernel might have to establish a temporary mapping to make both address spaces available at the same time.

Indirect strings avoid unnecessary copy operations to/from the message buffer. Up to 31 strings can be included in a memory message. On the receiver side, buffers for such strings can be specified so that the IPC can copy directly from server object to client object or vice versa. Scatter/gather permits strings to be gathered on the sender side and/or scattered on the receiver side. Thus multiple blocks can be directly transferred to a single receive buffer; a single send buffer can be split into multiple blocks. Figure 1 illustrates how a complex memory message is transferred.

Map messages map pages or larger parts of the sender's address space into the receiver's space. This feature enables user-level pagers and main-memory management on top of the microkernel. Special communication mechanisms based on shared regions can also be constructed.

2.2 SawMill

IBM's *SawMill* project aims at addressing the complexity of developing and maintaining a variety of custom operating systems. With the emergence of embedded and personal systems, the need to create operating systems customized to device and application requirements has increased significantly. The development and maintenance of these operating systems is quite unwieldy. As a first step, the *SawMill* project is developing an approach and tools to decompose existing operating systems into flexibly reusable components. The next step is to define an architecture upon which efficient and robust operating systems can be composed. This framework is being applied to Linux to create *SawMill* Linux which consists of Linux-based components running on top of the L4 microkernel. It provides typical system services through multiple user-level servers, such as file systems, device drivers and network systems. Further general components such as memory servers, task servers, and access control managers enable the composition of a coherent Linux system.

2.3 Flick

IDL Compilers such as Flick [4] are relatively easy to port to a new OS or middleware kernel, and they are extensible through new data types. The output of an IDL compiler is typically used as input for a general-purpose compiler, e.g. *gcc*, that a programmer uses for code development. Easy adaptation of the IDL compiler to new general-purpose compilers is a further relevant property.

Flick tries to generate efficient stub code by using inline functions and macros for the generated stubs whenever possible. Nevertheless, at least when combined with *gcc*, this results in huge amounts of data transfer operations that are logically superfluous. In theory, a compiler should be able to remove all of them. In practice, the required data flow analysis is too complicated; consequently, inefficient code is generated.

3 Designing a Stub Model

3.1 A Simple Stub Model

We first describe a simple stub model to illustrate the tasks performed by stub code on the client side and on the server side. For this simple model, we assume that a client invokes a procedure or method M that is supplied by the server. Synchronization and concurrency are ignored in this simple model. M has *in* parameters (values passed from the client to the server), *out* parameters (result values passed from the server back to the client), and *inout* parameters that are first passed to the server and then overwritten by results coming back from the server.

The IDL compiler generates a client stub procedure M_{client} for each function M in the interface definition. The client stub is called locally by the client application. The fact remains hidden that the service function does not run locally, but rather in another address space or even on another computer thousands of miles away. The client stub assembles a message with all the information the server requires to complete the task, including all the parameters (*marshalling*).

The message is then sent to the server, and the client waits for the server's reply. The reply message contains all out and inout result values. The client stub unpacks these values from the message and stores them in the appropriate client parameters (*unmarshalling*). In detail, the client stub M_{client} —

- C1 constructs a *request* message that contains all input and inout parameters, and a *key* that identifies the procedure/method M (*marshalling*);
- C2 sends the request message to the server that implements M and waits for a reply message from the server;
- C3 fills the inout and out parameters with data received through the reply message (*unmarhalling*); and
- C4 returns to the invoking client.

The server programmer implements a procedure M_{server} on the server side for each method M of the interface definition.

The IDL compiler generates a central code pattern that handles communication, decoding, marshalling, and unmarshalling of parameters. This central server code typically includes a main loop that receives requests from clients and distributes them to the corresponding server procedures M_{server}. For each M_{server}, the IDL compiler generates a server stub that examines the request packet and retrieves the input data (*unmarshalling*). The stub then invokes the routine itself and finally creates a reply for the client.

An IDL compiler should generate both the main loop and the stubs automatically. Users should be able to easily modify the loop code, because they might want to implement additional features, e.g. load balancing.

In detail, a thread that waits for client requests —

- S0 receives the request message and uses the received key to determine which procedure/method M should be invoked and which parameters are expected and will be returned by M;
- S1 extracts in and inout parameters from the received request message (*unmarshalling*);
- S2 calls the server procedure M_{server} with the extracted parameter values;
- S3 constructs a *reply* message and stores the result values of all inout and output parameters of procedure M_{server} in that message (*marshalling*);
- S4 sends the reply message back to the client.

Steps C2, S0, and S4 are basically determined by the underlying IPC system, in our case by the L4 microkernel. Steps C4 and S2 are determined by the general-purpose compiler used, in our case *gcc*. Marshalling and unmarshalling, steps C1+S1 and S3+C3, are less restricted and more crucial. As our experience with Flick shows, a less optimal model can easily result in significant copy overhead for marshalling and unmarshalling.

3.2 Marshalling Through Direct Stack Transfer

To get an idea of how parameters can be communicated most efficiently between M_{client} and M_{server}, we first look at a local procedure call. *Gcc* and many other C compilers push input-parameter values on the stack prior to procedure invocation. Figure 2 shows the stack layout for a procedure called with 3 input parameters. Now look at the remote case. Three parameter values have just been pushed on to the client stack (left, M_{client}). On the server side (right), M_{server} would ideally expect a stack of exactly the same content since

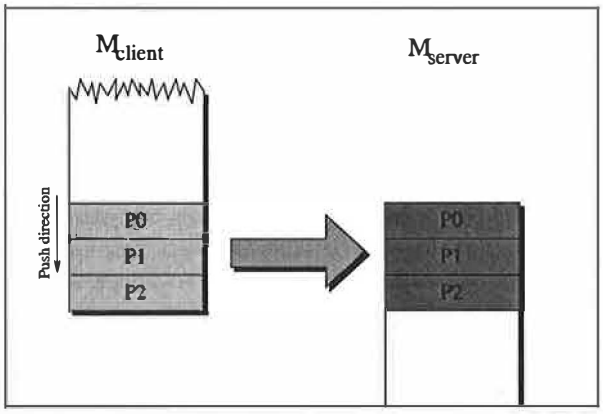


Figure 2. Procedure with 3 input parameters.

M_{server} has exactly the same parameters as M_{client}. Basically, the stub code had to copy the stack frame one-to-one from the client to the server stack. No additional operations would be required for parameter marshalling/unmarshalling.

Since out parameters in C are typically implemented through pointers (which are passed as in parameters), we have to extend the parameter set by pointers that point to those variables that are later sent back to the client as out parameters. Figure 3 illustrates the three basic layouts:

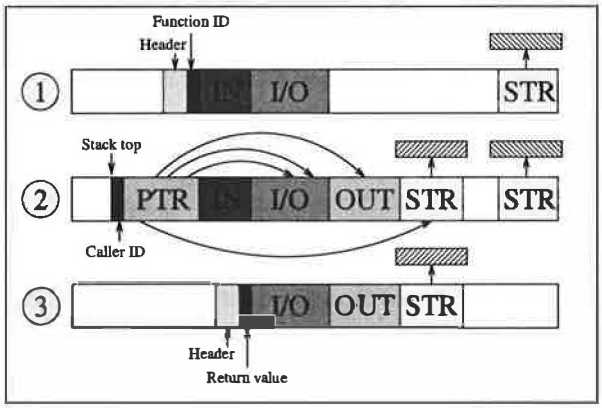


Figure 3. Message layouts. *(1): sent by the client to the server; (2): received message, extended to server stack; (3): message sent back by the server to the client.*

1. The client constructs a message that contains all in and inout values (plus optional strings). The message buffer has enough space to receive the reply message from the server.
2. The server extends the received message by pointers that make the inout and out parameters (and optional strings) accessible for the server procedure M_{server}. Then it invokes M_{server}. As a normal C function, it works on its input parameters (PTR, IN and the caller ID).
3. After returning from M_{server}, the stub removes pointer and in parameters from the stack, pushes the return value and an appropriate message header, and sends the resulting reply message to the client.

An immediate consequence of the stack and message layouts is that the IDL compiler must sort parameters to enforce the sequence in, inout, out.[1]

3.3 Complex Data Types

At this time, the only data types IDL[4] handles are 32-bit words and strings (up to 2 MB). It will be extended by pages to also handle mapping through IDL functions. Any other data type can be implemented through those basic types. Large objects like arrays or structs can be transferred as strings, while small objects (characters, short integers) may safely be extended to 32-bit words.

Extending smaller objects to words has no additional costs since *gcc* maps such objects to words anyhow when generating local function calls. Implementing large data types as indirect strings is beneficial since it avoids copying them into the message buffer.

4 Generated Code — An Example

To further illustrate details, we analyze the output that the compiler generates for the function *pfs_write* of the physical file system (pfs):

```
int pfs_write([in] int handle,
              [in, out] int *pos,
              [in] int len,
              [in] int data_size,
              [in, size_is(data_size)] int *data);
```

IDL[4] generates three files which contain the client stubs, the server stubs and the main server loop. Client and server stubs are generated as `asm` functions for *gcc*. The server loop is in C so that it can easily be modified by an application programmer. It is common to all functions and decodes incoming requests, i.e. selects the appropriate server function and invokes it through the server stub:

```
setupNewBuffer();
ipcReceive();
do {
    unpackQuery();
    callStub();
    packResponse();
    setupNewBuffer();
    ipcReplyWait();
  } while (1);
```

[1] A similar sorting mechanism is used to collect string parameters and pages to be mapped.

Client stub

Table 1 shows the output IDL[4] creates for the pfs_write() call on the client side. Assuming it hands over two parameters in registers, this stub consists of 17 instructions. In detail, the code sections (referring to the numbers in the code) work as follows:

1. *Create descriptors for indirect strings.* pfs_write() has one input string, `*data`, so a descriptor has to be created.
2. *Marshal parameters.* The input and inout parameters are pushed on the stack; inout parameters go first. Note that the last two parameters (`len` and `handl`) are not pushed, but loaded into the `EBX` and `EDI` registers.
3. *Generate message header.* The header specifies the number of dwords to be transferred for both directions, as well as one dword for the mapping function, which is not used here.
4. *Load registers for IPC and supply function key.* IDL[4] needs to specify the send and receive buffer addresses and a timeout. The function key is transferred via registers and loaded here as well.
5. *Invoke IPC call.*
6. *Unmarshal server output.* In the case of pfs_write(), a return value and the `*pos` parameter must be handled. These can be transferred via registers, so the memory buffer is entirely discarded.

Server stub

The stub (see Table 2) is called from the server loop. It converts the request message from the client into a stack frame for the server function:

1. *Move the stack pointer* to the message buffer. The message header and the function ID (which is the first dword in the payload) can be overwritten, so the new `ESP` points to the fifth dword in the buffer.
2. *Add pointers to strings and output values.* First, a pointer to `*pos` is pushed, then one to the input string buffer. Finally, the ID of the source thread is supplied.
3. *Perform function call.*
4. *Create reply message.* The input values and pointers are discarded, then the return value and a new message header are added.
5. *Restore the stack pointer.* Its original value was saved in `EBP` during the function call, as it is the only register that is automatically saved by *gcc*.

```
__inline__ extern sdword pfs_write(
   sm_service_t __service, sdword handl,
   sdword *pos, sdword len,
   sdword data_size, sdword *data)

{
   dword __return;

   int dummy0,dummy1,dummy2,dummy3;

   asm volatile (sub    $8, %esp);
   asm volatile (pushl  %0 ::"g" ((int)data));   // (1) push in string
   asm volatile (pushl  %0 ::"g" (data_size)); //     descriptor

   asm volatile (pushl  %0 ::"g" (*pos));        // (2) push 2 in
   asm volatile (pushl  %0 ::"g" (data_size)); //     parameters

   asm volatile (
         sub     $12, %%esp

         pushl   $0xA100             // (3) msg header,
         pushl   $0x8000             //     bits describe msg struct
         pushl   $0

         mov     %%esp, %%eax        // (4) ipc register setup
         pushl   %%ebp               //     save frame prt reg
         xor     %%ebp, %%ebp        //     reply msg type = short
         mov     func_id, %%edx      //     function key
         xor     %%ecx, %%ecx        //     timeouts = infinity

         int     $0x30 // (5)

         popl    %%ebp               // (6) (restore frame ptr reg)
         add     $48, %%esp          //     release stack space

   : "=S" (dummy0), "=d" (__return),
     "=b" (*pos), "=D" (dummy3)
   : "S" (__service), "D" (len),
     "b" (handl)
   : "%eax", "%ecx"
);

return __return;
}
```

Table 1. *Client stub for* pfs_write.

5 Performance

5.1 Measurement Environment — SawMill Linux

IDL[4] is used in the *SawMill* project for component communications. *SawMill* Linux is a Linux-derived multiserver OS where physical file systems (PFS), file and buffer cache, device drivers, network stack, VM subsystems such as anonymous memory, etc. are all implemented as user-level servers that communicate through L4 IPC and IDL[4] stubs.

For *SawMill*, we analyze the stubs that are required to let a normal Linux process execute file-system operations such as *open, read,* and *write*. The physical file system we used in the experiments is compatible to

```
__inline__ extern void *call_pfs_write(void *buf,
        int com_source, int *strlist)

{
   int __return,dummy0,dummy1;

   asm volatile (
                pushl %%ebp                 // (1)
                mov  %%esp, %%ebp
                mov  %%eax, %%esp

                mov  %%eax, %%edi           // (2)
                add  $12, %%edi
                pushl %%edi
                pushl 4(%%esi)
                pushl %%ebx

                call  _pfs_write            // (3)

                add  $24, %%esp             // (4)
                pushl %%eax
                pushl $0x2000
                pushl $0x2000
                pushl $0

                mov  %%esp, %%eax           // (5)
                mov  %%ebp, %%esp
                popl  %%ebp

        : "=a" (__return), "=b" (dummy0),
          "=S" (dummy1)
        : "a" (buf), "b" (com_source),
          "S" ((int)strlist)
        : "%ecx", "%edx", "%edi"
   );

   return (void*)__return;
}
```

Table 2. *Server stub for* pfs_write.

Linux' *ext2*. In fact, the *ext2* code was extracted from Linux and then combined with IDL^4-generated server templates. The resulting ext2-compliant PFS runs as a user-level server in its own address space. Libraries have been modified such that now IDL^4 stubs and L4 IPC communicate with *SawMill* servers. An *open* request is always sent first to the virtual file system (VFS) which propagates it to the corresponding PFS server. Subsequent *read/write* requests, however, are handled through direct communication between the user application and that PFS server, i.e., need only one RPC (two IPCs).

The normal *SawMill* Linux has all stubs generated by the IDL^4 compiler. In addition, we generated a second version of *SawMill* Linux whose stubs were all generated by the Flick compiler. For both versions, we measured stub instructions and application performance.

For our measurements, we used a Pentium III running at 500 MHz with 64 MB of main memory and a 540 MB IDE disk drive (IBM DALA-3540).

5.2 Effects On IOzone Throughput

The IOzone benchmark [1] begins by writing a file of 64kB, then it reads the contents twice. In the second read phase, all requests can be backed by the page cache. The performance of the second phase is completely determined by processor operations, basically for communication and for copying data into the user program's buffer, and not by disk accesses.

We measured reread throughput where IOzone read 4 KB[2] of file data per read request. Table 3 presents the overall performance results reported by IOzone (ten consecutive iterations). IDL^4 improves the IOzone throughput by approximately 13%. The time for a 4-KB read request decreases from 8.0 μs to 7.0 μs. Since reread costs are dominated by the data copy costs this result can only be explained by significant improvements in stub code.

IOzone reread throughput on *SawMill* Linux using	
Flick stubs	**IDL^4 stubs**
503 kB/s ± *17 kB/s*	**569 kB/s** ± *18 kB/s* *(+13%)*

Table 3. *Overall throughput (± standard deviation) in the IOzone benchmark.*

5.3 Stub-Code Instructions

To analyze the stub-code performance, we counted the executed instructions for the Flick-generated and the IDL^4-generated stubs. Table 4 compares the results for three *SawMill* file-system functions, *pfs_open, pfs_write,* and *pfs_get_direntries.* The numbers include all instructions that are executed in stubs and in the central server loop. For comparison, the number of instructions the L4 microkernel executes for the corresponding IPCs is also included. (Note that complex operations such as block transfer operations are counted per iteration.) The effective communication costs are then given by adding the stub costs —either Flick or IDL^4— to the IPC costs.

[2]Longer read requests effectively decrease application performance, independently of whether pure monolithic Linux or *SawMill*/Flick or *SawMill*/IDL^4 is used: The Pentium L1 cache has a size of 16 KB. If, e.g., 8 KB of data are copied from the page cache to the user buffer, this operation already floods the entire cache. So every other application or file system data access leads at first to a cache miss. Furthermore, since some further cache lines are also used for the data copy, the first part of the user buffer will be flushed from L1 at the end of the copy operation. Effectively, most application accesses to the data read will thus also lead to L1 cache miss except if a clever application would read its data or if the OS would copy its data in reverse direction.

int **pfs_open** ([in] int client, fobj, flags, mode, [out] int *handle)			
	IPC (kernel)	**Flick stub**	**IDL^4 stub**
client → server	*163*	116	65 *(–44%)*
client ← server	*95*	105	37 *(–61%)*
total	*258*	221	102 *(–54%)*
eff. comm. instructions, *IPCs+stubs*		**479**	**360** *(–25%)*

int **pfs_write** ([in] int handle, [in,out] *pos, [in] int len, data_size, [in, size_is data_size] int *data)			
	IPC (kernel)	**Flick stub**	**IDL^4 stub**
client → server	*248*	150	73 *(–51%)*
client ← server	*95*	105	38 *(–64%)*
total	*343*	255	111 *(–56%)*
eff. comm. instructions, *IPCs+stubs*		**598**	**454** *(–24%)*

int **pfs_get_direntries** ([in] int handle, [in,out] *pos, [in] int count, [out] int data_size, [out, size_is data_size] int **data)			
	IPC (kernel)	**Flick stub**	**IDL^4 stub**
client → server	*157*	145	79 *(–46%)*
client ← server	*248*	140	42 *(–70%)*
total	*405*	285	121 *(–58%)*
eff. comm. instructions, *IPCs+stubs*		**690**	**526** *(–24%)*

Table 4. *Instructions executed for Flick and* IDL^4 *stubs (client+server). The* IPC *column shows the instructions executed by the microkernel per IPC (this depends on message type and size). The* **effective communication instructions** *are the sum of the required* IPC *(kernel) instructions* plus *the (user) instructions of the stubs.*

Flick stubs take almost as many instructions as the microkernel needs for the IPC system call (including the message copy). Current IDL^4 stubs use only half as many instructions.

6 Portability Versus Specialization

The IDL^4 experiment gave us some evidence that specialization in stub-code generation pays and is perhaps even necessary for industrial acceptance of component-based system construction. However, the obvious questions are (1) how portable can an optimizing IDL code generator be made, and (2) what efforts are required to port a specific code generator to a different compiler or machine architecture?

Currently, the IDL^4 code generation is specialized for the *gcc* compiler and x86 processors. From our current experience, we can give some raw estimates about the costs to adapt IDL^4 to other architectures:

New register link conditions: Low adaptation costs, comparable to those that are required to modify the C bindings for all 7 microkernel system calls.

Different Processor: Low adaptation costs as long as the stack layout is similar. Basically, the stub templates used by the IDL^4 code generator have to be translated into the new machine/assembler language.

Different stack layout: Depending on how different the stack layout is, adaptation costs might be lower or higher. Different orderings or distances on the stack are easy; a runtime model without a stack might require designing a new data model for cross-address space parameter transfer.

Different C compiler: Easy if the C compiler offers *inline asm* procedures exactly like *gcc*. Medium-high costs if the compiler offers basically the same features but uses different syntax. Impossible or ineffective if the compiler offers no such features.

The last point is probably the most critical one. Optimization is hard or even impossible if the C compiler does not offer access to its code generation process. However, this seems to be an inherent problem of separating the IDL and the programming language. In all other cases, the adaptation costs are similar if not lower than porting a normal compiler.

7 Conclusions

IDL^4 shows that efficient stub code can be generated with reasonable effort. With the availability of fast IPC, the gains achievable through optimized stub code are becoming relevant for component-based systems. Multi-server operating systems can probably not be built efficiently without such optimized stubs.

We have shown that significant performance improvements are possible. Nevertheless, it is still open, how far the current IDL^4-generated stubs are from the optimum.

The optimized stub-code generation requires specialization of the IDL compiler's code generator in two dimensions, firstly, toward the target programming language and compiler, secondly, toward the target machine. In this area, two questions are still open: (1) How specialized (with respect to acceptable efficiency) must an optimizing IDL compiler be? (2) Can we find a small set of templates and/or methods that permit easy and low-cost specialization of an optimizing IDL compiler

for most existing programming-language compilers and hardware architectures?

An obvious next step therefore is to determine whether and how the current results can be generalized. An ideal solution would permit extension of the portable Flick compiler with the presented code-generation techniques.

References

[1] *The IOZone filesystem benchmark*, April 2000. Available from http://www.iozone.org/.

[2] J. Bruno, J. Brustoloni, E. Gabber, A. Silberschatz, and C. Small. Pebble: A component-based operating system for embedded applications. *Proc. USENIX Workshop on Embedded Systems*, pages 55–65, 1999.

[3] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, 1998.

[4] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstorm. Flick: A flexible, optimizing idl compiler. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, pages 44–56, June 1997.

[5] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The SawMill multi-server approach. In 9^{th} *ACM SIGOPS European Workshop*, Koldingfjord, Denmark, September 2000.

[6] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, St. Malo, oct 1997.

[7] The Object Management Group (OMG). *The Complete CORBAServices Book*. http://www.omg.org/library/csindx.html.

HP Caliper – An Architecture for Performance Analysis Tools

Robert Hundt
Hewlett-Packard Company
robert_hundt@hp.com

Abstract

HP Caliper is an architecture for software developer tools that deal with executable (binary) programs. It provides a common framework that allows building of a wide variety of tools for doing performance analysis, profiling, coverage analysis, correctness checking, and testing. HP Caliper uses a technology known as dynamic instrumentation, which allows program instructions to be changed on-the-fly with instrumentation probes. Dynamic instrumentation makes HP Caliper easy to use: It requires no special preparation of an application, supports shared libraries, collects data for multiple threads, and has low intrusion and overhead. This paper describes HP Caliper for HP-UX, running on the IA-64 (Itanium) processor. It describes Caliper's architecture, dynamic instrumentation algorithm, and the experiences gathered during its implementation.

1. Introduction

The IA-64 processor's instruction set architecture (ISA) offers an impressive set of architectural features which explicitly create synergy between compilers and the processor [10]. The IA-64 groups up to three instructions in bundles for execution in parallel and can issue multiple bundles per clock. The architecture provides 128 integer registers, 128 floating point registers, 64 1-bit predicate registers, and 8 branch registers. Both control and data speculation are supported, as well as predication to eliminate branches, software pipelining of loops, and branch prediction.

These processor features enable powerful program optimizations. However, their efficiency depends on the dynamic run-time behavior of a given program, which can only be guessed by a static compiler. Additionally, modern software paradigms emphasize distributed systems, component-based modularization and object-oriented designs. This further prevents compilers from optimization and analysis on a global scope.

Over the last years, the computing community has developed a strong set of tools and methods used to analyze and monitor run-time behavior of a program. Statistical sampling and binary instrumentation are two of the major techniques.

Statistical sampling is typically performed by taking periodic snapshots of the program state, e.g., its instruction pointer (IP). Sampling is considered to be light-weight, non-intrusive, and imprecise. It imposes low overhead on a program's run-time performance and can be used for time-critical experiments. However, measurements are statistical samples and have errors. Without special hardware support, due to super-scalar issues, deep pipelining, and out-of-order instruction completion, a sampled IP may not be related to the instruction address that caused a particular sampling event. Some architectures introduce a varying offset to the IP at a particular sampling event [2, 10].

The IA-64's performance measurement unit (PMU) offers programmable CPU event counters, event address registers (EAR), and a branch trace buffer (BTB). The PMU supports a set of over 150 event types, allowing a wide range of system analysis tasks [10], such as analysis of cache misses, translation look-aside buffer (TLB) misses, or instruction cycles. When such a hardware counter overflows, it is possible to precisely link events to an instruction address with help of the event address registers (EAR).

Dynamic binary instrumentation allows program instructions to be changed on-the-fly and leads to a whole class of more precise results. Measurements such as basic-block coverage and function invocation counting are accurate. Since the binary code of a program is modified, all interactions with the processor and operating system may change significantly, for example a program's cache and paging behavior. Instrumentation is therefore considered to be intrusive. Due to additional instructions, execution time can slow down anywhere from some percent to factors like 2x or 4x. Dynamic instrumentation, as opposed to static instrumentation, is performed at run-time of a program and only instruments those parts of an executable that are actually executed. This minimizes the overhead imposed by the instrumentation process itself.

Tools based on dynamic instrumentation require no special preparation of an executable, like many other tools for performance analysis and tuning do. Such treatment could be recompilation with a special compiler flag, or a modified link process before or during program start. A good example is profile-based optimization (PBO). There, a program must be recompiled with a special flag to insert counting code in the program and to output a trace profile at the end of the program run. Feeding this profile back

into the compiler allows combining of static analysis and runtime information and to generate a highly optimized application for a representative set of input data. This data combination also requires another compiler flag to be used. PBO generates efficient code, but is complicated to use, especially for large-scale software systems. It has not been widely accepted by the software industry.

HP Caliper (or *Caliper* for short) integrates PMU supported sampling and fast dynamic instrumentation. It offers a framework for performance analysis tools for binary executables and requires no special preparation or recompilation of these binaries. It supports shared libraries, collects data for multiple threads and processes, and has low intrusion and overhead. This paper describes HP Caliper for HP-UX, running on the IA-64 (Itanium) processor. It describes HP Caliper's architecture and public interfaces, presents the dynamic instrumentation algorithm and details experiences gathered and lessons learned during its implementation.

2. Related Work

This section describes related work as characterized by Cmelik and Keppel [5]. They present a list of over 45 hardware emulators, "decode-and-dispatch" interpreters, "pre-decode" interpreters working on intermediate representations, static cross compilers, and dynamic cross compilers. These tools differ in support for kernel code, time of instrumentation, requirements for debug information, and support for signals and multithreaded programs.

Many tools try to generalize static or dynamic instrumentation and create abstractions of machines, file formats, compiler code layouts and optimization strategies. These tools often come with additional generators for machine abstractions.

Paradyn [9] is a performance measurement tool for parallel and distributed programs. It includes an abstract, machine independent, dynamic instrumentation API (DynInst), and provides precise performance data down to the procedure level.

The Parallel Tools Consortium sponsors two related projects, the Performance API (PAPI) project and the Dynamic Probe Class Library (DPCL), the latter being based on Paradyn.

Spike [6] is a profile-directed optimization system. It uses code-layout to improve cache behavior and hot-cold optimization to minimize the number of instructions executed on frequent paths through a program. Atom (Analysis Tools with OM) is a tool based on Om [14]. Atom NT is a set of tools built with the Spike library, including profilers, arc counters, and simulators for cache and branch prediction units.

Some tools and libraries allow static instrumentation of binary executables. EEL [11] is a machine-independent library for editing executables and provides abstractions which allow tools to analyze and modify binary executables. Etch [13] is a tool which allows instrumentation of Win32/Intel executables. Tools based on Etch include call graph profilers and instruction execution analyzers. UQBT [4] is a retargetable and "resourceable" binary translator. Resourceable means that it accepts a binary from one of several platforms as input, which is then transformed to an intermediate representation and finally retargeted to several target machines.

Rational's Purify, Quantify, and PureCoverage [12] are systems which perform static instrumentation for error detection, run-time performance analysis, and coverage analysis. Intel's Vtune is a low-level CPU sampler which allows detection of CPU bottlenecks and cache behavior.

HP's Aries [18] combines fast code interpretation with dynamic translation in order to execute PA-RISC applications transparently and accurately on IA-64 systems running HP-UX.

Previous work at HP's dynamic instrumentation lab includes a callback driven dynamic instrumentation environment and dynamic optimizers for x86, PA-RISC, and IA64. A transparent dynamic optimizer named Dynamo is under development at the HP Laboratory in Cambridge.

3. HP Caliper Architecture

HP Caliper is physically organized as a shared object library with the Caliper API as its interface. A tool built with HP Caliper runs as a *Developer Tool Process*, controlling an *Application Process* via the operating system's debug interface (e.g., ttrace on HP-UX or /proc on Linux).

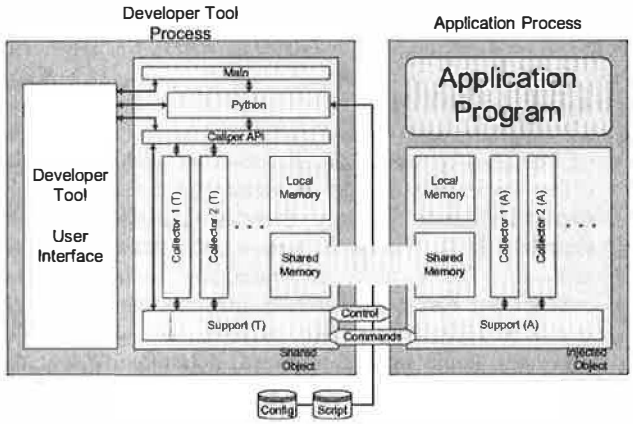


Fig 1: HP Caliper Architecture

Developer tools based on HP Caliper are physically split into two parts, the user interface and the HP Caliper shared object (libcaliper.so). User interfaces can be stand-alone scripts or integrated development environments (IDE). HP Caliper allows to inject an optional run-time library into the application process to record information,

react on application events, and communicate with the developer tool.

The shared object is HP Caliper's main component. It contains support code, collectors, the Caliper API, and memory management routines. It integrates a Python interpreter and provides a default C main function.

The HP Caliper API consists of a set of C function interfaces to the main architectural blocks of HP Caliper. The interfaces, although written in ANSI C, follow object-oriented design principles and form a simple object model consisting of Measurement sets, Events, Processes, Configuration, Context and Collectors. These classes are described in the following paragraphs.

Measurement sets enable measurement specification and combination. Instrumentation-based measurements include function coverage and counting, basic block coverage and counting, arc counts, and call graphs. PMU-based global performance metrics include control speculation miss ratio, data speculation miss ratio, ALAT capacity miss ratio, data and instruction cache miss ratios, TLB miss ratios, and more. Statistics of branch mispredictions and branch taken ratios can be obtained.

Event objects deal with application and user events and handle event queues. Typical program events include process creation and destruction, shared object loading and unloading, timer expiration, PMU counter overflow, and process termination.

Process is a set of interfaces allowing creation of or attachment to a process as well as handling of process related events, such as signals. It allows controlling processes via the OS's debug interface (e.g., ttrace or /proc).

Configuration permits to parameterize HP Caliper and to set parameters such as initial size of shared memory blocks.

Context allows HP Caliper to scale to large applications by narrowing down measurements in both time and space. A context's three dimensions are:

- *Address* - to include or exclude modules (DLLs), functions and address ranges
- *Time* - to schedule measurements
- *Event* - to specify program actions for specific program events (e.g., fork / exec).

A *Collector* is a tool built into HP Caliper that performs a special kind of measurement, for example, PMU sampling or instrumentation-based function counting. Collectors use the infrastructure offered by HP Caliper. Each collector adds an individual API to the HP Caliper API to interact with the developer tool. On the application side, support code for the instrumentation may be injected and each individual collector may inject additional private code. Data and control transfers between HP Caliper and an application use shared memory.

The *Caliper Support Library* offers a framework of services and tools for dynamic instrumentation and sampling. These services include:

- encoding and decoding of machine instructions to an intermediate representation (IR) with automatic fix-up of IP-relative branches.
- handling of an executable's ELF file, code and data segments, debug and unwind information, and function tables.
- managing data exchange between HP Caliper and its monitored processes (e.g., for counters, events, or control instructions).
- controlling a process with the debug and performance measurement interfaces (perfmon()).

A developer tool communicates with HP Caliper via the Caliper API or via the integrated Python interpreter. This Python interpreter performs multiple tasks. It contains wrappers for all API functions and is used to interpret initialization and configuration scripts. The interpreter acts as the main interface for all command line tools and as the main shell for the integrated debugger cdb (described later). It can also be directly accessed from the graphical user interface and from the C main function. Currently, Python can not be used to describe probe code sequences at a high level.

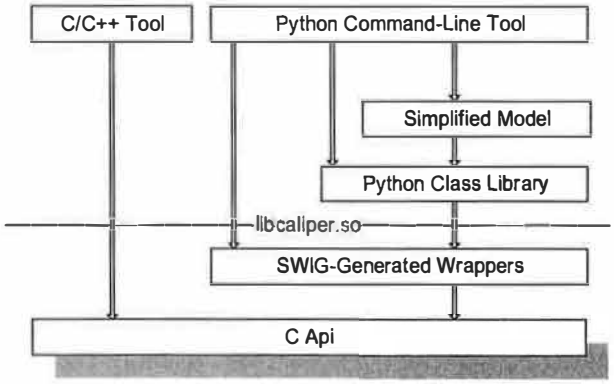
The API resides in a set of C header files, which are processed by SWIG [16] to generate Python wrapper code. The interpreter and the wrappers are included in the HP Caliper shared object. SWIG uses text templates to generate code and some templates had to be changed to make SWIG usable on a 64-bit processor with "new" data types like uint64_t.

The generated wrapper code is very complex to use. Therefore, a Python class library was developed based on the SWIG-generated interfaces. These classes are more intuitive and serve as the main scripting interface to HP Caliper.

This programming model was also felt to be too complex for simple and standardized tasks. This was especially true for novice users, since knowledge of the Caliper API and its object model was required. A further simplified model was developed which only considers the most basic user control requirements. In this model, only a few variables can be assigned before a measurement starts, and all other details are hidden. For example, these variables include the name of the application to be monitored and the type of measurement to be performed.

Tools using HP Caliper can access the C API, the Python SWIG-generated functions, the Python class library, the simplified layer or operate HP Caliper from an IDE. Other language interfaces, such as C++ or Java, can be added on top of the C API.

A small and simple driver is sufficient in order to perform useful work with the HP Caliper shared object. All such a command-line driver does is analyzing its parameters in order to find arguments specifying a script file and finally running this script.



An arc counter standalone tool is written in roughly 20 lines of Python code. (The following code snippet is simplified for clarity):

```
#!./caliper
import caliper, os, sys
try:
# Create process and load executable
    test_exec = os.path.abspath(sys.argv[1])
    proc      = caliper.process()

    proc.load(test_exec,
              sys.argv[1:],
              ["PATH=."])

# Create context
    context = caliper.context(proc)

# Create collector: arc_counter
    arc_count = caliper.arc_count(context)

# Run the measurement
    establish_measurement()

# Retrieve counters and generate report
    fout = open(test_exec + ".pbo", "w");
    arc_count.report(fout)
    fout.close()
except:
    ...
```

A control file in the simplified model looks like:

```
# specify applicatioin
application = "a.out"

# specify output file
pbo_out     = "flow.dat"

# run collector
collect(pbo)
```

4. Dynamic Instrumentation

This section explains HP Caliper's dynamic instrumentation algorithm. It briefly discusses, why it can be characterized as a lazy algorithm before it finally outlines the experiences gathered during its implementation and testing.

One of the major benefits of dynamic instrumentation, as opposed to static instrumentation, is scalability. According to the 80:20 rule (in a typical program, 80 percent of the runtime is spent in 20 percent of the code), only a small fraction of an executable system has to be instrumented in order to detect the most significant parts of a program.

Dynamic instrumentation can be performed in a variety of ways. The two strategies we considered for generating probe code were to either make use of trampolines (out-of-line), or inline and relocate probe code (in-line).

As an example, an out-of-line instrumentation strategy may perform code transformations like the following in order to perform function counting. A given function foo's entry point may look like this in IA-64 assembly:

```
foo::
  alloc         r33=ar.pfs,0,11,1,0
  addl          r9=-2944,r1
  addl          r8=-2936,r1
foo'::
  . . .
```

The out-of-line strategy will instrument foo's entry point with a long branch to a trampoline that executes the original instruction, plus some additional code to update an invocation counter.

```
foo::
  nop.m
  brl   trampoline
foo'::
  . . .

trampoline::
// save state, create free register rx
  . . .

// execute original instructions
  alloc         r33=ar.pfs,0,11,1,0
  addl          r9=-2944,r1
  addl          r8=-2936,r1


// perform additional tasks
// update a counter for this function

  movl rx, addr-of-counter
  fetchadd [rx], 1

// restore state
  . . .

// return to original code.
  brl  foo'
```

There are, of course, many possibilities for encoding, reaching, and returning from the actual trampoline code. Care must be taken for code with branch instructions in the first bundle of a function. Trampoline code and original code may be farther apart than the 25-bit encoded relative address offsets of the IA-64 allow. Therefore, long branches have to be used.

This strategy has several advantages. If all of the probe code is placed out-of-line and the instrumented instructions branch to it and back, then the counting code will not cause any wrinkles in the address space of the original application. Thus, all branches would continue to reach their designated targets. It is also easy to combine multiple instrumentations simply by cascading trampolines.

The other major strategy for probe code generation is to inline and relocate code. The above code snippet for function foo would then be transformed into the following:

```
foo_instrumented::

// modified alloc instruction to
// generate free register
  alloc          r33=ar.pfs,0,12,1,0

// perform additional tasks,
// update a counter for this function

  movl r45, addr-of-counter
  fetchadd [r45], 1

  addl           r9=-2944,r1
  addl           r8=-2936,r1
```

This strategy leads to more compact code, less intrusion, and better performance. It does, however, come at a price.

Insertion of probe code changes the relative offsets in a code stream and requires lookup of indirect branches (in a translation table) whose target cannot be determined by the instrumenter. Combining different instrumentations and probe code is not as easy as it is in the well-defined, sand-box style trampoline approach.

Susan L. Graham, et. al. [8] investigated the relative overhead associated with the inline and out-of-line instrumentation strategies and found the overhead to be 34% for inline and 112% for out-of-line strategies. The transformation overhead is computed as the run-time of all code that is added to the application in order to support the primary probe code, without including the probe code itself. The benchmark included spec programs such as compress, gcc, li, sc, espresso, and more.

The use of long branches had to be minimized for another reason. The first versions of the IA-64 only emulate the long branch, which causes additional run-time performance impacts. A trampoline-based instrumentation approach with out-of-line branches made heavy use of long branches and was therefore disregarded in favor of the current in-line approach.

Preliminary measurements on HP-UX showed that the overhead of a long call branch, compared to a short call branch, is approximately 100 to 300 cycles. This number was considered to be small for an emulated instruction and permitted us to use the long branch instruction "occasionally" in the algorithm.

The inlining relocation method is faster even without considering the extra cost of an emulated long branch instruction. This justified our algorithmic decision in the light of an upcoming, hardware-supported long branch instruction.

4.1 Algorithm

HP Caliper's approach works at the granularity level of functions, which are always instrumented as a whole. Probes are inlined into functions and instrumented functions are relocated.

The dynamic instrumentation algorithm performs the following five steps, which are encapsulated in the Caliper API:

1. *Attach and Inject*: HP Caliper identifies an executable or an already running process. It attaches to a process using the HP-UX ttrace system call. The process stops and transfers control to HP Caliper, which injects code into the process which allocates shared memory and optionally adds run-time libraries for dynamic instrumentation.

2. *Function Discovery*: Function entry points are identified by analysis of the unwind information tables (sometimes called exception tables), the procedure lookup tables, and the symbol table. Unlike a debugger, HP Caliper does not depend on debug information in order to perform this step. The analysis may still miss some function entry points because of a lack of unwind information and symbolic information. However, these functions are discovered dynamically. Whenever a call target cannot be found in the internal function dictionary during instrumentation, a break is inserted at the target address of a call branch, assuming it to be a function entry point.

3. *Static Break Insertion*: Every function's entry point is patched with a break instruction.

4. *Run under Dynamic Instrumentation:* Control is transferred back to the process. The process runs until it hits one of the inserted break instructions at the entry point of a function. Since the process is controlled by ttrace, control transfers to HP Caliper and the instrumentation process begins at the current function.

The function is analyzed for instrumentability, probe codes are inlined into the function, IP-relative references are updated, counters are created, and an instrumented version of the function is moved to shared memory. The original function's entry point is patched with a long branch instruction to its instrumented version. Break instructions are inserted at function external IP-relative branches, whose targets have not yet been instrumented or have not been identified by function discovery.

After instrumentation, control transfers to the instrumented function, which continues to run until it hits the next break instruction. Control will again transfer to HP Caliper and the dynamic instrumentation process is resumed.

5. *Output*: Upon process termination or user request, control again transfers to HP Caliper. Statistics, counters, and other measurement results are now retrieved and

output into one of the integrated, collector-specific formats or via user-defined, script-based output routines.

This dynamic instrumentation algorithm could rightfully be characterized as a lazy instrumentation algorithm. If a program were to consist of only one, presumably huge, function *F*, the algorithm would instrument the whole program at once after reaching *F*'s entry point. No code transformations that depend on information only available at run-time are performed.

Programs consisting of only one function, however, are not a standard case in today's computing environments, right the opposite is true! The instrumentation sequence also depends on the dynamic control flow of a program and can be changed interactively or via the definition of context. We, therefore, continue to use the term "dynamic instrumentation" to describe this algorithm.

Without further explanation we would like to mention another property of this algorithm, the possibility to mix instrumented and non-instrumented code without hurting program correctness.

4.2 Experiences

This section describes the most important experiences gathered and lessons learned during implementation and debugging of this algorithm for HP-UX.

The IA-64 contains a high performance register stack engine (RSE) which helps to minimize the cost of creating a call frame and a function call by maintaining a separate register stack. If a programming model requires consistent unwinding of the stack, e.g. during a C++ exception, both program stack and register stack have to be unwound.

For every region in a program, unwind information is generated and stored in the text segment for fast access. The presence of unwind information is a requirement by the IA-64 runtime software architecture [10]. If code motion happens during instrumentation, the unwind information must be dynamically updated. This is no easy task, since regions get modified by probe code inlining. Unwind information updating is not yet fully resolved and blocks HP Caliper from being used for analysis of C++ programs which make use of C++ exceptions.

Compilers frequently translate a C/C++ switch statement into an indirect branch based on a branch table located in the code segment. HP's compilers place branch tables in a read-only data section of the text segment. It is generally impossible for a binary code analyzer to decide whether a given address contains data (such as an entry of a branch table) or real code. Some algorithms exist to identify branch tables and, for some code generation schemes, this problem can always be solved [3,17].

HP Caliper uses compiler-generated annotations residing in an executable's ELF file to identify these tables. If a branch table has been identified, the table entries are patched so they point to their corresponding instrumented target addresses. Whenever an indirect branch is executed based on an unmodified branch table, it will go to a function's non-instrumented version. Although the program maintains correctness, the resulting counter values become imprecise.

Because of this, HP Caliper's precision depends on the presence of annotations. Annotations are linked to the unwind information for a given region and as long as the executable contains unwind information, the corresponding annotations can be found.

The foremost requirement for binary instrumentation is, of course, to preserve the program semantics at any given time. Probe code needs free registers and earlier approaches required the compiler to reserve registers for special use of a post link-time tools. This is again a topic of a recent discussions in the industry and certainly is a most convenient approach. However, the compiler group soon experienced register pressure and, consequently, this approach was skipped.

HP Caliper now uses a staged method to find free registers. Free registers are first identified with the assistance of compiler generated annotations. If no annotations are found, free registers are created by increasing the number of stacked output registers of a function by modifying a function's dominating alloc instruction. This will fail if the function doesn't have an alloc instruction, has multiple alloc instructions or because the alloc instruction already allocates all the stacked registers. In such a case, explicitly spilling/filling to the program stack is necessary.

Multithreaded applications presented a new kind of challenge for HP Caliper. Insertion of an instruction (e.g., a long branch instruction), turned out to be more complicated than expected. The IA-64 bundle size is 16 bytes, but load and store instructions only operate on a maximum of 8 bytes. This means that two store instructions are necessary to update a bundle. In multithreaded applications, there are two potentially hazardous scenarios. It is possible that a thread could hit a bundle while being in the middle of it's update process, thus executing a half-deployed instruction with an invalid instruction template field, which will result in a signal. Or, a thread could have been stalled on slot 1 or slot 2 of a bundle, waking up on a changed instruction, again resulting in incorrect program behavior.

The latter scenario has been solved in HP Caliper using a sequence of update steps. The first problem requires the installation of a signal handler for invalid template exceptions. This has not been implemented yet.

To date, HP Caliper simply halts all threads in the target application while performing an instruction update. While guaranteeing correct program behavior, this method slows down execution speed, especially on multi-processor systems. For this reason it will later be changed to a method with full support for multithreading.

The IA-64 supports call shadows where two branches are located in one bundle as in this example:

```
     nop.m     0
(p6) br.call.dptk.few  .-0x150
     br.call.sptk.few  .-0x410;; // shadowed
```

If the predicate register p6 is set to 1 then the first branch instruction is executed. Since branch targets and return address are always full bundle addresses on the IA-64, the second branch will never be executed.

If this instruction sequence is instrumented and counting code is inserted, then the original instructions get dispersed across multiple bundles, changing the implicit logic of the call shadow that supressed the second branch. Thus, HP Caliper performs an additional search for call shadows and alters the instrumentation sequences accordingly.

Break instructions are used in a similar, but conflicting way by both HP Caliper and debuggers, making it impossible to debug a HP Caliper-controlled application. Therefore we integrated debugging functionality into HP Caliper and named the driver for this functionality cdb (Caliper Debugger). This feature became invaluable for identifying program flaws, invalid probe code sequences, kernel bugs and forgotten stop bits all over the instrumented code.

Cdb makes use of the integrated Python interpreter to display a prompt, to parse commands, and to perform actions accordingly. It supports insertion of break points, single stepping, disassembly of original and instrumented code, dumping of data and registers, and more. HP Caliper allows falling back to a cdb prompt whenever an unexpected situation or signal occurs.

Scripting languages such as Python typically have powerful support for socket communications of some kind. It was easy for us to offer a remote interface to cdb. This proved to be valuable during debugging of applications which expect input from stdin via redirection. The implementation of this remote functionality is concise and simple.

HP Caliper also has some limitations. Instrumentation does not work with dynamically generated code, with programs that internally change between little-endian and big-endian or with programs that use IP-aware signal handlers. It is also possible to create assembler code sequences where instrumentation will fail, for example code performing label arithmetic. However, such sequences are rarely used, if at all.

At the time of this writing, HP Caliper is able to successfully instrument the first ten Spec2000 benchmark programs (164.gzip, 175.vpr, 181.mcf, 197.parser, 168.wupwise and more) to perform function coverage analysis, function counting and arc counting on IP-relative call branches as well as hazard checking for predicates

4.3 Case Study: Predicate Hazard Checking

Predicate hazard checking is an interesting application of the HP Caliper framework and is presented here as a case study.

The HP compiler optimization group developed an algorithm where instructions are placed in the same issue group, although they may have a resource conflict, as in the following bundle with a read-after-write conflict:

```
      nop.m
(p35) add1  r14=0x40784634,r0  // write r14
(p36) ld4.s r15=[r14]          // read  r14
```

The instructions, however, are predicated. If it can be guaranteed that the predicates are never 1 at the same time, then this is a powerful optimization technique.

In order to verify the algorithm, the optimization group uses a static tool to read in ELF executables and to output potential hazards as tuples <hazard address, predicate register, predicate register>. Hundreds and thousands of potential hazards are indicated by the static tool. This information is then manually checked against disassembled code and run through other static analyzers.

Still, there had to be some form of dynamic verification. If one single occurring hazard was found, it was proven that the algorithm had a flaw for a given input stream.

In order to support our compiler optimization team, we wrote a collector which reads in the output of the static hazard analyzer and instruments functions containing potential hazards. The probe code sequences check whether or not two indicated predicate registers are both set to one at a questionable address and increase a counter for this hazardous case. If a single counter has a value of one or greater, an actual hazard has been found.

Implementing this collector was a straightforward operation, because all major building blocks like counter management, function discovery, probe code generation, insertion and handling of break instruction were already in place. The tasks to perform for hazard checking were more or less to define an input format and reader for the hazard file and the layout and implementation of the probe code sequences. We have been able to identify hazards and helped the optimizer group to improve their algorithms.

5. Future Work

HP Caliper will be ported to Linux on IA64 processors and to HP-UX on PA-RISC.

What are the IA-64 specific features used by HP Caliper that will complicate porting it to PA-RISC? The ISA of both processors is fairly similar and the success of HP's Aries emulator running PA-RISC applications on IA-64 demonstrates this. There are however two main problem areas:

- There is no PMU or equivalent hardware on PA-RISC. It is therefore expected that HP Caliper for PA-RISC will focus on binary code instrumentation.
- HP Caliper exploits two instructions unique to the IA-64, the long branch instruction `brl` and the memory access synchronizing `fetchadd` instruction for counter updates. For both instructions there is no equivalent on PA-RISC,

and workarounds for their usage must be developed.

No problems caused by operating system dependencies are expected. Although the debug and perfmon interfaces of HP-UX and Linux differ, their capabilities are both similar and powerful enough to allow HP Caliper to be ported

In order to fully support analysis of C++ programs making use of exceptions, the dynamic updating of unwind information will be developed soon.

An optional *Caliper Agent* above the Caliper API is under development. This agent routes API calls between the developer tool and the HP Caliper shared object, enabling a HP Caliper for distributed systems. The agent uses remote procedure calls (RPC) based on code generated from Caliper's API header files.

More tools will be developed on top of the instrumentation framework. In particular, these will include basic block related tools and API checkers such as a memory leak detection tool and a pthread correctness checker. Caliper's design will also change slightly to enable dynamic loading of collectors.

One of the more interesting challenges for the future is dynamic code transformation, e.g., optimization. Light-weight sampling will identify hot traces and dynamic instrumentation will optimize a program using this information. The optimizations may further adapt themselves as the characteristics of input data sets change.

6. Acknowledgements

HP Caliper is the result of a strong team effort within HP. Thanks to Dave Babcock, Eric Gouriou and German Rivera for their contributions. Special thanks to Umesh Krishnaswamy, Vinodha Ramasamy and Thomas Lofgren for their additional feedback and help during my writing of this paper.

I would also like to thank the anonymous reviewers for their invaluable feedback.

References

[1] A. Aho, R. Sethi, and J. Ullman, Compilers: Principles, Techniques, and Tools (Mass.: Addison-Wesley, 1985).

[2] J. Anderson et al., "Continuous Profiling: Where Have All the Cycles Gone?" Proceedings of the Sixteenth ACM Symposium on Operating System Principles, Saint-Malo, France (October 1997): 1–14.

[3] Chrstina Cifuentes, Antoine Fraboulet "Intraprocedural Slicing of Binary Executables", University of Queensland, Australia.

[4] Chrstina Cifuentes, Mike van Emmerik "UQBT – A Resourceable and Retargetable Binary Translator", University of Queensland, Australia (December 1999)

[5] Robert F. Cmelik and David Keppel, "Shade: A Fast Instruction-Set Simulator for Execution Profiling", Sun Micosystems Laboratories, Incorporated, and the University of Washington, technical report SMLI 93-12 and UWCSE 93-06-06, 1993

[6] R. Cohn, D. Goodwin, P. G. Lowney, "Optimizing Alpha Executables on Windows NT with Spike", Digital Technical Journal 9, 4

[7] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin, "Spike: An Optimizer for Alpha/NT Executables," The USENIX Windows NT Workshop Proceedings, Seattle, Wash. (August 1997): 17–24.

[8] Susan L. Graham, Steven Lucco, Robert Wahbe, "Adaptable Binary Programs", Usenix 1995

[9] Jeffrey K. Hollingsworth, Barton P. Miller, "Dynamic Instrumentation API", Journal, University of Wisconsin, 1996.

[10] Intel Corperation "IA-64 Application Developer's Architecture Guide", May 1999

[11] J.R. Larus and E. Schnarr "EEL: Machine Independent Executable Editing. In SIGPLAN Conference on Programming Languages, Design and Implementation, pages 291-300, June 1995

[12] Rational Software Cooperation. Product documentation for Purify, Quantify and PureCoverage.

[13] Ted Romer et al. "Instrumentation and Optimization of Win32/Intel Executables using Etch", Usenix Windows NT Workshop 1997

[14] A. Srivastava and David W. Wall. A Practical System for Intermodule Code Optimization at Link-Time. Journal of Programming Language, 1(1), pp 1-18, March 1993.

[15] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation, Orlando, Fla. (June 1994): 196–205.

[16] SWIG – Simplified Wrapper and Interface Generator, Dave Beazley et al., University of Utah, Open Source Project at http://www.swig.org

[17] M. Weiser "Program Slicing". IEEE Transactions on Software Engineering, SE-10(4):352-257, July 1984

[18] Cindy Zheng, Carol Thompson "PA-RISC to IA-64: Transparent Execution, No Recompilation", IEEE Computer Society Cover Feature, 3/2000

Incremental Linking on HP-UX

Dmitry Mikulin
Hewlett-Packard Company
mikulin@cup.hp.com

Murali Vijayasundaram
Hewlett-Packard Company
vm@cup.hp.com

Loreena Wong
Hewlett-Packard Company
loreena@cup.hp.com

Abstract

The linker is often a time bottleneck in the development of large applications. Traditional linkers process all input files, even if only one or two objects have changed since the previous link. To shorten link time, we have developed an incremental linker for HP-UX which only processes modified files. Users can take advantage of the performance gains without modifying their usage patterns of the existing HP-UX linker since the incremental linker is implemented on top of the regular 64-bit linker. In addition to the tasks of the normal linker, the incremental linker must save extra information about input files, symbols and relocations, allow for the expansion of existing files and addition of new ones by allocating padding spaces in the output file and use this information to perform in-place updates. The results of several different design considerations and tradeoffs are materialized in link-time performance gains of up to thirteen times that of a normal link for large applications.

1. Introduction

1.1 Motivation behind incremental linking

In recent years application sizes have grown dramatically. This increase has been enabled by significant advances in software development tools such as object-oriented languages which shorten development cycles and allow smaller groups of developers to produce very complex applications with hundreds of thousands or even millions of lines of code. Understandably, these applications take longer to compile and link. This increase becomes most obvious during application debugging when multiple edit-compile-link-debug cycles must be performed in a short period of time. Most changes involve only a small portion of an application source base, typically one or two modules. The problem of long compile times is partially solved by 'make' utilities that recompile only modified source files. Unfortunately there is no easy solution for traditional linkers—they still need to process all object files and libraries to resolve cross-module references and assign addresses to all symbols. The linker becomes a bottleneck that significantly affects a developer's productivity.

The only way to shorten link time, apart from making algorithmic improvements in the linker itself, is to reduce the size of input processed by the linker. Since only a few object files are typically modified before a link is performed, it is possible to reduce the input size by only processing modified objects and replacing their contribution to the output files in place. This fact is the motivation behind the incremental linker.

An incremental linker should process only those input files that have been modified since the last time a link was performed. With meticulous planning and saving of necessary data, all other input files that have not changed should not need to be reprocessed.

1.2 Related work

Despite the great importance of fast link times and the fact that several production systems have implemented incremental linkers (IBM mainframes had incremental linking capabilities since the 1960s; Sun Microsystems offers this functionality which can be optionally enabled on their Solaris systems; Microsoft C/C++ Development Studio has it enabled by default in debug links), very little has been written on the subject.

An academic paper by Quong and Linton [1] provided us with valuable analysis of padding space allocation and reuse. It also contained important performance data for programs of various sizes. Even though the authors chose a different approach in their implementation (a memory resident component which maintains state information about the incremental linker), the paper gave us insight into usage patterns as well as performance and implementation trade-offs.

A paper by Hoffman and Curwen [2] described an alternative way of solving the problem of long link times based on dynamic linking. Object files

comprising the application are grouped into shared libraries which are smaller and therefore can be linked faster. Source changes cause only enclosing libraries to be rebuilt. This approach primarily addresses linking simple applications and will fail in cases with name collisions between shared and archive libraries. Dynamic linking is not always supported (e.g. in embedded systems) and may require significant build process changes which makes the scheme not very practical.

1.3 The HP-UX standard linker

The main job of the linker is to merge several relocatable object files into a single load module (a shared library or an executable program file). In doing this merge, the linker must resolve references across the input objects, layout and assign addresses to the resulting load module. In addition, the linker must be able to handle references to symbols defined outside the current load module. For example, if several object files are linked together with a shared library, the referenced code in the shared library is not copied into the output file. Instead, the linker creates dynamic symbols for these types of references and places information in the output file so that when the proper shared library is brought in at runtime by the dynamic loader, the symbol reference can be bound based on the load address of the program file and shared library. Dynamic symbols are also created for all global symbols which may be referenced outside the load module.

Another key task of the linker is to perform symbol resolution—the process of matching references to the definition of a symbol. If a reference can be bound to a definition within the load module, the linker can simply replace the reference with the address of the symbol. If the reference can not be bound to a definition within the load module, the linker will create a dynamic relocation. The dynamic relocations will be processed by the dynamic loader at runtime.

The linker also supports a –r option to merge multiple object files into a single relocatable object. When the –r option is specified, the linker will retain the symbol and relocation information in the output file, making it suitable for subsequent re-linking.

2. Overview of incremental linking

Incremental linking support on HP-UX is implemented as part of the standard 64-bit linker. Aside from a few exceptions, the vast majority of linker options and functionality is available with incremental linking as well, enabling users to take advantage of the performance gains without sacrificing linker functionality when building shared libraries and executables.

The 64-bit linker now has three operating modes:

- Normal link mode: Normal linker operation. This is the default mode.
- Initial incremental link mode: This mode is entered when the +ild option is specified and the output load module (executable or shared library) does not exist or the output load module is not an incrementally linked executable. In this mode, the linker will create an output load module that is suitable for incremental linking.
- Incremental link mode: This mode is entered when the +ild options is specified and the incrementally linked output load module exists. In this mode, the linker will make incremental updates to the output load module. It is in this mode that the great performance gains can be realized.

This paper focuses only on the latter two modes.

In initial incremental links, the linker processes all input object files and libraries the same as it does for normal links. In addition to this basic functionality, the incremental linker must do additional work to enable subsequent incremental links. The linker must store extra information about the input files processed, all global symbols, as well as relocations. Also, the incremental linker must allocate proper padding space for text, data, bss, and other sections in the output file in order to allow room for future expansion from additional input files, definitions, references, etc. Because of this added functionality required for initial incremental links, the time spent in initial links is slightly higher than that of a normal link.

Once all the proper information has been stored in an initial incremental link, subsequent incremental links can be performed. The linker uses timestamps on individual input files to determine which files have changed and only reprocesses modified files during incremental links. The linker uses saved relocation information to patch the symbolic references in the rest of the output file. These tasks are described in greater detail in the rest of this paper.

The incremental linker is intended for use by programmers during developmental stages only, and should not be used for release builds of products. Because the incremental linker pads sections for future expansion, programs are bloated in size and contain information not necessary for execution of the program. In addition, the incremental linker is incompatible with most compile- and link-time optimization techniques and thus cannot produce optimized executables. The incremental linker, however, is an excellent timesaving tool for use during development when programmers are constantly adding

and modifying small portions of code and rebuilding programs for testing. In addition, incrementally linked programs are still source-level debuggable in the same manner as normally linked executables.

The remainder of this document describes design considerations and implementation details, and performance results of the incremental linker for HP-UX.

3. Command line, file and library processing

Because our incremental linker is implemented as an extension of the regular system linker, practically all options are allowed when building incrementally linked executables and shared libraries. The use of the mapfile option, which allow users to change the layout of the output load module through a mapping file, is also supported, as long as the specified mapfile doesn't change between incremental links; if it does, we fall back to an initial incremental link. The only options incompatible with incremental linking are link time performance and optimization options. Some linker options significantly influence the resulting output module and if added to or removed from the command line, may make the executable difficult or impossible to incrementally update. For this reason, we decided not to allow any command line option changes in incremental links with the exception of tracing and verbose options; these can be freely added, removed or changed since they have no effect on the resulting output file.

In an initial incremental link, file names, types and time stamps for all object files, shared libraries, archive libraries and their processed members are saved in the output file. In subsequent incremental links, this information is used to detect which files need to be reprocessed. For object files, the logic is simple: if a file's time stamp has changed, that file needs to be reprocessed.

If an archive library is modified, we need to check each individual member processed in previous links for time stamp changes and reprocess only the ones that were actually modified. In some cases, we may need to reprocess an archive library even if the archive itself was not modified. If any of the modified object files introduces a new unresolved symbol reference, we need to scan archive library symbol tables and extract members that define that symbol. However, contributions of archive members are never removed from the output file, even if there are no more references to any of the symbols defined in those members.

Since symbols from shared libraries are not propagated into the output file, we do not reprocess shared libraries during incremental links. Instead, it is the dynamic loader's responsibility to catch unresolved symbols at runtime.

4. Padding and reuse of space

The incremental linker re-links programs by inserting modified object code into the existing output file. During the initial incremental link various output file sections such as text, data, bss etc., are padded with additional space for future expansion. The output file data structures like symbol table, section table and linkage tables are also padded with additional space.

During incremental links, the linker will vacate the space occupied by modified object files. The vacated space in the output file will be reused when the contents of the modified object files are copied over to the output file. The incremental linker always tries to fit the modified object file's contributions into their previous location. After several incremental links, the padding space may become exhausted. When this occurs, the incremental linker will fall back to performing a full initial incremental link during which additional padding space will be allocated.

4.1 Padding space allocation

The incremental linker allocates two kinds of padding spaces:

- File specific padding space. Each section in the output file consists of contributions from all object files. The Figure 1 shows the layout of an output section created during normal link. The incremental linker allocates a padding space after contributions form each object file. These file specific padding spaces are allocated to allow for possible growth in the object file's contributions. Figure 2 shows the layout of an output section with file specific padding.
- Generic padding space. During incremental links new object files may be added to the link. To accommodate contributions from new object files, a large padding area (as shown in Figure 2) is allocated at the end of each output section.

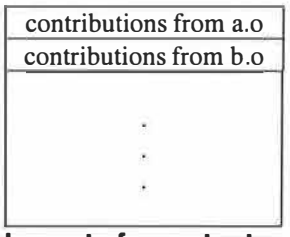


Figure 1. Layout of an output section by the normal linker

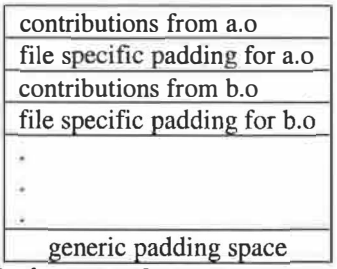


Figure 2. Layout of an output section by the incremental linker

4.2 Keeping track of section layout

To enable the incremental linker to efficiently locate and replace a modified object file's old contributions, we maintain a data structure called a linkmap in the output file. The linkmap consists of two sections: a file table and a section map.

- File table. The file table contains a mapping of file identifiers to path names of object files. The file identifier is a unique number assigned to each object file present in the link. The contributions from each object file are laid out in the output sections in sorted order based on the file identifiers of the object files.
- Section map. The section map records the layout of the output section. For each output section we maintain an array of section mappings. Each mapping consists of the following information:
  - File identifier of the contributing object file.
  - Section relative offset of the location where the object file's contributions are copied over.
  - Size of object file's contributions.
  - Size of file specific padding space.

  Just as the output section contents are sorted on file identifiers of contributing object files, the section mapping data is sorted on file identifiers.

4.3 Replacing old contributions

During incremental links we perform the following steps to copy contributions from modified object files to the output file:

- Locate the modified object file's old contributions in the output section by searching through the section mapping data. Since the section mappings are sorted on file identifiers, a binary search can be performed to locate the object file's contributions.
- Check if the object file's old contributions and file specific padding space is sufficient to fit the new contributions. If the contributions fit, they are copied into the existing space in the output file. The file specific padding increases the chance that the modified file's contributions will fit into their previous location.
- If the contributions no longer fit in their pervious location, fall back to a full initial incremental link.

If there are any new object files in the incremental link, their contributions are copied over to the generic padding area. We also assign increasing unique numeric values as file identifiers to new object files. This keeps the section layout sorted on file identifiers. The file table and section mapping data in the linkmap are consistently updated during incremental links.

5. Symbols

One of the main responsibilities of the linker is to maintain information about symbols, resolve undefined and multiply defined symbols, and assign symbol values. Even though basic rules to maintain both static (.symtab) and dynamic (.dynsym) symbol tables are the same, more information is needed to deal with symbol resolution and assigning symbol values in incremental links. The linker generates various data structures for every symbol—linkage table entries, dynamic relocations, etc. To avoid inefficient use of file space and reuse these structures in incremental links we need a means of keeping track of them. Also, resolving symbols with multiple definitions, some modified and others unchanged, is a non-trivial task. A 100% solution for these challenges would require a substantial amount of additional information to be saved, maintained, and processed on every link, which could negate all advantages of the incremental linker. We had to find a solution acceptable in a vast majority of cases, yet very lightweight both in terms of space and processing time.

5.1 Symbol table management

The existing symbol table structure intermingled local and global symbols without distinction as to which input file defined each symbol. For incremental links, that structure was not sufficient. We needed a structure that allowed us to modify only those entries in the symbol table that pertained to the modified file(s) in an incremental link. In addition, padding areas must be allocated in the correct places to allow for additional symbol definitions, if any, from modified files.

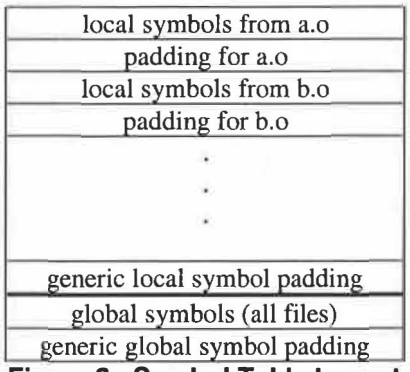


Figure 3. Symbol Table Layout

The symbol table structure is subdivided into two main pieces (Figure 3), one for local symbols and one for global symbols. In addition, there is a parallel secondary symbol table (.ild_symtab) which stores incremental linking-specific information about symbols. Since all local symbols for a given file will be reprocessed in incremental links, information regarding them is not stored in the .ild_symtab. Instead, only information about global symbols is stored there. The additional information is used mainly for symbol resolution in incremental links, which is described in the next section. During incremental links, the symbol tables are reconstructed from the original output file, and all modified global symbols are updated in place. Any new symbols can be added into the appropriate padding area. Once the link is complete, only the modified sections of the symbol tables are rewritten in the output file.

Unlike global symbols, local symbols are not reconstructed in incremental links. Since these symbols are local to the modified file only, the resolution rules are much less complex, and there is no reason to store any additional information about these symbols in the .ild_symtab. The local symbols section of the symbol table is sectioned by file identifier, one section per file. Each file-specific section for local symbols has its own padding space, and there is an additional generic padding section at the end of the locals for any additional files that may be added in incremental links (see Figure 3). The linkmap stores the mapping for the entire symbol table, including each file-specific local symbol section. During incremental links, the linkmap helps determine the symbol indexes of local symbol definitions for modified files and the local symbols are reprocessed, overwriting the old definitions. This is different from the way global symbols are processed in that they are not updated in place, but rather they are completely reprocessed.

5.2 Resolving symbols

The linker resolves symbols based on their definition types. This is a fairly easy task when there is only one definition for a particular symbol in a link—all references are resolved to that definition. However, symbol resolution becomes more complicated when a symbol has multiple definitions such as common (storage request), weak, and strong definitions. In a regular link a symbol with the strongest definition type 'wins' over symbols with lower precedence definition types. In incremental links only new and modified files are processed, which means that the linker only scans symbol tables from those files. Situations when only one object file contains a symbol definition are easy to handle. In incremental links we need to update the symbol definition in the output symbol table only if the object containing the definition was modified. If the symbol has multiple definitions it is sometimes difficult to pick the new winner. To illustrate this problem, we will look at a few examples.

Suppose we have two object files, a.o and b.o. One file (a.o) contains a storage request for 'data1' integer; the other (b.o) contains an initialized definition for 'data1'. In a regular link the definition from b.o wins symbol resolution. Suppose the user made changes to remove `data1' from b.o altogether. The regular linker would pick the definition from a.o as a winner. But in an incremental link, if a.o was not modified, the linker does not have enough information to pick the new winner. Ideally, for every symbol we would need to have a list of definitions we have seen in previous links so that we can find a new winning definition and resolve all references to the new symbol. This approach would require us to store quite a bit of additional symbol information in a form of symbol definition lists. These lists, like all other data structures, would need to be padded in case new symbol definitions are added, and updated in incremental links. This solution would add complexity to the implementation, requiring more storage and increasing processing time.

Instead we decided to go with a more lightweight solution which resolves symbols correctly in the majority of cases. In cases where the incremental linker cannot determine the new winner, it will fall back to an initial incremental link.

In an incremental link we maintain two copies of every symbol: the old winner— a winning definition from the previous link restored from the output file; the current winner— a winning definition we have seen so far while processing modified files. At the end of the first pass over modified objects, the linker has to make a decision for every symbol as to what version to pick as the new symbol definition (See Figure 4).

If the object containing the old definition was not modified we just follow regular logic to perform symbol resolution between old and current definitions. If the defining object was modified, there are two possibilities as to which should be the current winner with respect to the old winner. If both symbols come from the same object file, we pick the current winner. If they come from different files, we check which version of the symbol would win regular symbol resolution by the normal linker. If the current version wins, we have no problem; we just take it as the new winner. If the old version wins, we have to do an initial incremental link because we do not have enough information to declare any symbol a winner. If we had a complete list of symbol definitions from the previous link, we could find a symbol which would have won symbol resolution had the old winner not been there at all. But since we decided against maintaining this list, the only available option to recover is to fall back to a full initial incremental link.

```
if (old copy not modified) {
   do regular resolve(old, current)
}
else {
   if (old and current are
       from the same file) {
      resolve to current
   }
   else {
      if (current wins resolve(old,
                               current)){
         keep current as new winner
      }
      else {
         if (old and current are common){
            resolve to current
         }
         else {
            do initial incr. link
         }
      }
   }
}
```

Figure 4. Symbol resolution algorithm.

This may not sound like a good option since it may cause frequent re-links. However, it is not as bad as it seems. There are only a couple of ways to create multiple symbol definitions that do not cause a duplicate symbol error. One is to use common symbols; the other is to use weak definitions. System libraries primarily use weak symbols to prevent user name space pollution. Therefore the only way to get into a situation when the linker has to perform an initial incremental link caused by weak symbols is when a user removes a strong definition overriding a weak symbol from a system library. Under normal circumstances this change will not be performed very frequently. The situation is very similar with common symbols. If a user has multiple commons and an initialized (winning) definition for a symbol, and the winning definition is removed, fall back to the initial incremental link. Even though a change like this is likely to be more frequent, it will only account for a small percentage of code modifications and will not cause an overwhelming number of re-links.

One more corner case involves common symbols and can be handled without re-links. Suppose a user has multiple common definitions with the same name but different sizes. The definition with the biggest size wins symbol resolution. If in an incremental link the winning symbol is removed, we don't have to do a full initial incremental link. Since space has already been allocated, we can always pick the current winner even if it has smaller size.

5.3 Symbol resolution in libraries

The way shared and archive libraries are processed slightly changes the behavior of the linker with respect to reporting unresolved symbols. Since dependent shared libraries are not processed in incremental links, the linker is unable to tell whether a certain symbol is unresolved because there is no definition for it at all, or because its definition is in one of the dependent libraries. In the latter case, there is no problem; the dynamic loader will resolve the symbol at run time. To enable this dynamic symbol resolution, the incremental linker converts all potentially unresolved symbols into dynamic symbols during all dynamic links and lets the loader report errors in case these symbol cannot be found in any of the dependent libraries at runtime.

Because archive members are never removed from the link, run time behavior of some incrementally linked programs may differ from that of programs linked by a normal linker. Suppose you incrementally linked a shared library, liba.sl. One (and only one) of the objects (a.o) in the link referenced a global function func() which was resolved by an archive member func.o from lib.a. Now you remove the call to func() from a.o, so func.o is no longer needed. But since it is not removed in a subsequent incremental link, the symbol 'func' will remain exported by liba.sl and available for look-up. Thus, an application linked with liba.sl and referencing func() will still be able to successfully find it, even though it would have failed if liba.sl had been linked by a regular linker.

In our experience, the cases described above are not very frequent and there is a simple remedy to fix them – perform an initial incremental link.

6. C++ compile-time template instantiation

The HP-UX C++ compiler uses COMDAT to support compile-time template instantiation. COMDAT is a scheme that allows multiple, duplicate copies of code and data to be merged together by the linker into a single copy in the final executable. The comdat allows the compiler to generate multiple copies of a template function in separate object files. The linker must identify sections containing duplicate information and choose one of the copies for inclusion in the output file. The content of a COMDAT section is essentially a directory for the members of the COMDAT set. The section consists of an array of section indexes that point to the member section's entries in the object file section table. When object files containing COMDAT groups are linked, there may be more than one copy of a given COMDAT group. The linker chooses one of these copies to include in the final output file and discards the rest. If the same COMDAT group is defined in multiple files, they are assumed to be functionally equivalent.

In the context of incremental linking, the linker has to handle three different situations: modification of an object file containing a COMDAT group, addition of a new COMDAT group, and removal of a COMDAT group from an object file.

To support full functionality, the incremental linker has to keep track of the list of COMDAT groups contributed by each object file. For each COMDAT group in a modified object file, the incremental linker must check if the COMDAT group was chosen from this object file in the previous link. If so, it should invalidate the COMDAT group in the output file and replace it with the modified COMDAT group. When a new COMDAT group is added to an object file, the linker has to check whether the COMDAT group is already present in the output file. If it is, the COMDAT group should be invalidated and replaced with the new COMDAT group from the modified object file. If the COMDAT group is not already present, it should be added to the output file. Ideally when a COMDAT group is removed from a file, the linker should decide whether to physically remove the group from the output file or replace it from another object file. This scheme would require an extensive amount of bookkeeping. Also the additional information would need to be updated during incremental links. In a big C++ application there may be thousands of COMDAT groups. For example, in one of the test programs we analyzed, there were approximately 130,000 incoming COMDAT groups of which about 40,000 were unique. This scheme would greatly increase the complexity of the implementation, requiring more storage and potentially increasing the incremental link time.

Instead we decided to implement a simplified scheme that does not require maintaining any additional information. In our simplified scheme, the linker chooses a COMDAT group from one of the modified objects and discards the rest. If the COMDAT group is already present in the output file, it will be invalidated and updated with the new version. It is more difficult to handle the case when a COMDAT group is removed from an object that contributed it in the previous link. There are two cases to consider: either the same COMDAT group is defined in another object file or no other object file defines the same COMDAT group. Since we don't keep the list of all COMDAT groups defined by object files, we cannot determine whether the original COMDAT group has been replaced by the next available COMDAT group from a different file. In the second case, if the linker does not remove the COMDAT group, it is possible for the linker to miss a potential unsatisfied symbol error or report duplicate symbol definition that it wouldn't have in a normal link. Instead of implementing a complicated scheme and sacrificing performance, we fall back to a full initial incremental link when a COMDAT group is removed from an object file that contributed it in the previous link.

7. Linkage tables

Linkage tables (LTs) are generated by the linker to enable position independent code and data accesses. Our linker creates three kinds of linkage tables: PLT—procedure linkage table, DLT---data linkage table, and OPD—official procedure label descriptor table.

Each linkage table type is maintained in a similar fashion. As runtime components, LT entries are not allowed to change their position from one incremental link to another. Symbols contain indexes of these entries in order to have access to their corresponding LT entries. These indexes are assigned once for every symbol and after that never change in incremental links. It is very hard to come up with meaningful criteria to group LT entries. DLT and PLT entries are driven by symbol references. With multiple references in multiple objects, it is impossible to attribute an LT entry to any particular file. OPDs are driven by definitions, but in incremental links, definitions may move from object to object. So we decided not to group them at all. Instead we update them in place and rely on our underlying I/O buffering to capture any locality. Linkage tables only use generic padding for all new entries.

8. Import stubs

PLT entries are created in response to direct function calls that are potentially outside the load module we are currently building. These calls are redirected to an import stub that loads a procedure address and the target module's global pointer (GP) from a PLT entry and branches to that address. Calls to local functions can be relocated at link time and do not normally require import stubs. This means that in incremental links, all call sites to modified local functions would have to be relocated. To avoid saving all PC-relative relocations in an output file, we decided to use a different approach. In incremental links all PC-relative calls are directed to go through import stubs. This way we only need to update PLT entries for modified functions to insure that call site and target are connected correctly. One may argue that we added extra overhead for direct calls which slows down the application at run time, but since incremental linker is intended for debugging purposes only, runtime performance is rarely an issue.

Another problem we encountered was how to handle fixing up call sites to import stubs themselves. Normally, a single stub is created to service all PC-relative calls to a particular function. These stubs are attached to text contributions of objects for which they were generated. This makes them impossible to locate in incremental links. Also, if an object for which an import stub was created changes, we need to adjust references to this import stub for all unmodified files as well. This operation can be costly in terms of space (we would need to save information to keep track of import stubs) and, more importantly, link time (we would need to apply relocations for unmodified files). To make the operation simple and fast we decided to generate one import stub per symbol per input object file. The stubs are recreated in incremental links only for modified files and only direct calls from modified files need to be relocated.

9. Static and dynamic relocations

Another key linker functionality is to resolve external symbol references. In an object file these references are expressed as relocation records. Relocations from object files are processed and applied at link time if possible; if not, they are transformed into dynamic relocations applied by the dynamic loader at run time. Correctness of incremental links largely depends on our ability to maintain and process relocations.

Relocations can be broken up into several major categories. The first category is relocations applied to text sections. These relocations must be applied at link time and do not generate dynamic relocations because at run time text is not writable and cannot be relocated. The second category of relocations is those applied to linkage tables. These relocations are generated by the linker itself and may or may not require dynamic relocations depending on symbol types, output module type (executable or shared library), link type (static or dynamic), etc. And finally, there are relocations that are applied to data sections. These mostly deal with static data initializations and can generate dynamic relocations.

9.1 Text segment relocations

All relocations on text sections are in one way or another converted into linkage table relative relocations. Therefore if linkage table updates are done correctly, we do not need to worry about saving static relocations for these sections and reapplying them in incremental links. However, for error reporting purposes, we need to know what symbols were referred in all LT-relative relocations.

Suppose a.o defines a function foo() and b.o calls that function. If in an incremental link the definition is removed from a.o and b.o remains unchanged, we need a way to tell that foo() was referenced from b.o and issue an unresolved symbol message. If the definition for foo() was modified (it's address changed) we just need to update the OPD and the PLT entries for it. Since we don't actually need to apply relocations of this sort for unmodified files, we decided not to save them in their entirety, but rather save symbol indexes: one entry per referenced symbol per object file. In incremental links we scan entries from unmodified files and issue appropriate warnings if corresponding symbols are no longer defined. Unresolved references from modified objects will be detected as part of regular symbol and relocation processing. For locality, all entries are grouped by file and use both file specific and generic padding to accommodate expansion.

9.2 Linkage table relocations

Linkage table entries may require dynamic relocations. Changes of symbol attributes in incremental links may require new dynamic relocations to be generated for entries that did not have them in previous links. Some entries that had dynamic relocations in previous links may not need them any more. This means we must maintain a correspondence between linkage table entries and dynamic relocations for those entries to be able to perform updates. We decided that the easiest and the most efficient way to deal with this problem is to keep linkage tables and LT dynamic relocations in parallel tables. We avoid maintaining any additional information to help us find

relocations for LT entries as well as using potentially costly look-up schemes.

9.3 Data segment relocations

Unlike linkage table relocations, we could not avoid saving input relocation records for data segment relocations. We need to reapply these relocations in incremental links for symbols that were modified even though files containing these relocations remained unchanged. We also need to update dynamic relocations if symbol attributes change. For these cases, we extended the standard relocation record to contain an index of the corresponding dynamic relocation. These relocations are grouped by file for better locality of updates and use both file specific and generic padding to accommodate expansion. In incremental links, all records from unmodified objects are scanned. If a symbol is modified, the relocation is applied; a corresponding dynamic relocation is updated if needed.

9.4 Runtime behavior

One of the initial design requirements of the incremental linker was to ensure that the runtime behavior of programs remained the same for incrementally linked programs versus normally linked programs. This meant that there could be no changes to the format, layout, and semantics of any dynamic structures, including dynamic relocations. But as mentioned earlier, we needed to pad dynamic relocation sections for expansion. Also, in incremental links, symbol attribute changes may no longer require dynamic relocations for structures that required them in previous links; these relocations have to be wiped out. We had to use meaningful relocations that would be understood by the dynamic loader without changing the runtime behavior of programs. Consequently, in incremental links we create a dummy common symbol and fill all padding areas with relocations for this symbol. As a result, application start-up time is slightly slower, but since the target use of the incremental linker is for debugging, runtime performance is rarely an issue.

10. Performance

We measured the performance of the incremental linker using the following four programs of varying sizes: bison, the GNU parser generator; gcc, the GNU C compiler, and a large C++ customer application.

Table 1 shows the parameter for each of these sample test programs. All measurements were taken on a HP N4000 server with eight PA-8500 440Mhz CPUs. The system has 16 gigabytes of RAM and runs HP-UX 11.00 operating system.

	# of objects	Source language	#of symbols	Size of output file (Mb)
bison	**23**	**C**	**517**	**0.24**
gcc	**261**	**C**	**7423**	**2.93**
customer program	**1717**	**C++**	**389444**	**118.71**

Table 1. Sample Test Programs

	Normal link	Initial incremental link		Incremental link	
	sec	sec	% of normal link	sec	#times faster than normal link
bison	**0.5**	**0.6**	**120**	**0.15**	**3.3**
gcc	**3.5**	**4.1**	**117**	**0.41**	**8.5**
customer program	**131.0**	**158.0**	**120**	**11.70**	**11.2**

Table 2. Link Time Measurements

Table 2 shows the link time data. We measured and compared the normal and initial incremental link times. The initial incremental link is generally slower than the normal link due to the extra work done in these links. For each of the sample programs we measured, we found that the initial incremental link takes about 17% to 20% more time than a normal link with the same sources.

We next measured the incremental link times after making changes to two object files. For large programs, the incremental link is on average ten times faster than the normal link. The time taken by the incremental linker depends on the amount of code modified. Regardless of the amount of code modified, the time spent on extracting information from the output file will always be the same. We measured the link time on the largest test program (customer program)—after changing up to 30 large object members in an archive library. The link times are shown in Table 3. Even when large number of objects are modified, the incremental link is significantly faster than the normal link. For example, re-linking after changing 30 objects takes about 16.7 seconds which it about 8 times faster than the normal link.

#of objects changed	Incremental link time (sec)
2	**11.3**
4	**11.8**
6	**13.0**
8	**13.1**
10	**13.9**
30	**16.7**

Table 3. Incremental link times

The Table 4 shows the increase in size of the sample programs due to incremental linking. The size increase is due to two reasons: additional incremental linking data stored in the executable and the padding space.

	% increase in size	% increase due to padding
bison	**185**	**152**
gcc	**76**	**65**
customer program	**57**	**29**

Table 4. Program size increase

11. Current status and future work

The implementation was completed only a few months ago and the general HP-UX developer community hasn't had a chance to use the incremental linking capabilities. However, a few internal partners successfully used the incremental linker and provided us with valuable feedback on its correctness, usabily and performance.

Even though a tremendous amount of work has been done to design and implement the incremental linker, there are still a few areas that could use improvement. Currently our linker does not handle removal of symbols and objects very well. Symbols are never removed from either the static or dynamic symbol tables. This may cause undefined behavior of dynamic programs which reference removed symbols. Also, removal of object files from the link line is not handled the way it should be and causes an initial incremental link. Shared libraries are not processed at all in incremental links. Even though this reduces link time, the runtime behavior of programs may be confusing for less advanced users and they might want to have an option for the incremental linker to handle shared libraries the way regular linkers do.

Currently we don't handle padding space very efficiently, particularly in the area of accommodating file expansion. When a file's contribution to an output section uses up all file specific padding, we do a full initial incremental link even though there may be enough space in the generic padding area to store it.

The linker has all the information it needs to deal with these situations, and we will implement these improvements in the near future to reduce the number of limitations and differences between applications and libraries produced by the regular and the incremental linkers. These enhancements will also improve efficiency of file space reuse and shorten incremental link time.

12. Conclusions

We have developed an incremental linker for HP-UX that significantly shortens the edit-compile-link-debug cycle by substantially improving the link time. For large programs, our incremental linker is an order of magnitude faster than the normal linker is, allowing developers faster turnaround after simple bug fixes.

In our design we have chosen to forgo complex schemes that require vast amounts of bookkeeping in an attempt to guarantee incremental linking 100 percent of the time. Instead, we chose lightweight schemes that address the majority of situations that occur in practice. When rare corner cases are encountered, we fall back to performing a full initial incremental link. This choice has significantly reduced the complexity of the incremental linker and maintains the excellent performance gains achieved by the incremental linker. While there are still some enhancements that can be made, the incremental linker has already proven to be an extremely useful developer's tool with significant performance gains.

References

[1] Linking Programs Incrementally. Russel W. Quong, Mark A. Linton. ACM Transactions on Programming Languages and Systems, Vol. 13, No. 1. January 1991.

[2] Pseudo-Incremental Linking for C/C++. William A. Hoffman, Rupert W. Curwen. Dr. Dobb's Journal, October 1999.

[3] 64-Bit Run-Time Architecture for PA-RISC 2.0 http://www.software.hp.com/STK/partner/pa64rt.pdf

[4] ELF-64 Object File Format http://www.software.hp.com/STK/partner/elf-64-hp.pdf

[5] HP-UX Linker and Libraries User's Guide http://docs.hp.com/dynaweb/hpux11/dtdccn1a/lnkuen1a

Automatic Precompiled Headers: Speeding up C++ Application Build Times

Tara Krishnaswamy
Internet & IA-64 Foundations Lab, HP
tara@cup.hp.com

Abstract

This paper describes the crucial design and implementation issues that arise in building a fully automatic precompiled header mechanism for compiling industrial-strength C and C++ applications. The key challenges include designing the Makefile-transparent automation, determining the precompile-able region, capturing the compile environment and verifying it and addressing the correctness issues involved in using precompiled headers. The ensuing discussion treats the internals of the actual dumping and loading of precompiled headers as a black-box beyond a brief high-level description. An automatic precompiled header mechanism has been implemented in aCC, the HP ANSI C++ compiler, and the results of compiling real applications show that it achieves significant speedup in compile-times of real applications.

Introduction

C++ compilers are increasingly called upon to translate large applications that implement formidable computing tasks. These applications range from enterprise level software for supply-chain planning and database management to technical computing systems for CAD, mechanical design and automation and internet software for web browsing. These large software systems are structured as various C++ programs and distributed between many directories in the file-system. Each set of programs that constitutes these applications is compiled and linked into shared or archive libraries.

Also, such large systems often contain a mix of existing C and C++ code with incremental additions of fresh code for each new release of the product. As the size of these applications increases with each release, the time required to build them also increases, thereby impacting the productivity of the developers. In order to minimize this impact, many industrial systems initiate complete builds at night and require such builds to finish overnight, in spite of the applications' already large and steadily increasing size. These end-to-end builds include both the compile and the link phases, but are dominated by the compile phase.

At the source level, these large programs are modularized in traditional fashion as source-files and header-files. Source files primarily contain code that implements the behavior of the application, while header files mostly contain declarations that describe various complex types and data-structures used in the application. Such applications use both system headers that advertise the underlying operating-system API and application-specific headers that describe user-defined types and data-structures.

These header files are then programmatically internalized into one or more source files, exposing the types and data-structure interfaces to the source-files that need them. This header-file inclusion mechanism also allows the compiler to perform any static type checking mandated by the C/C++ languages. This paradigm of modularizing programs attempts to separate the interface to various data-structures and types in the application from the code that uses and manipulates it.

However, neither the C nor C++ language definitions enforce this paradigm strictly, allowing practically any legal C/C++ construct to reside in header-files. This complicates the structure of the application since C header files often contain not only type declarations but also macro definitions. Macro definitions associate symbol names, possibly with arguments, with replacement text which can contain conditional statements, loops, function calls and other executable code. C++ header files add further complexity since they contain both the interfaces to various classes and the definitions of the inline member functions of those classes.

In essence, the C++ promise of software reuse with libraries is delivered to a large extent via static header files that contain a lot of code that needs to be recompiled each time the importing translation unit is compiled. This proliferation of header files with complex code demands huge compiler effort to digest and thereby negatively impacts the compile-times of the applications. Table 1 contrasts the volume of code imported by a C++ *"Hello World!"* program against one

written in C. Table 2 compares the volume of code between source files and header files for some applications. The table shows that on average about 87% of the application's code resides in header files.

From Table 2 it is apparent that in order to achieve fast and efficient compiles, a compiler must seek to minimize header-processing time. Note that although about 87% of lines in the applications stem from header files, it does not imply that an equivalent portion of compile-time will be dedicated to header processing. This is partly because header files predominantly contain declarations that don't trigger machine code generation, although debug-mode compiles may cause debug information to be emitted for these declarations. Table 3 reports the compile-time division between actual source line processing and header processing.

Note that the processing times for the header files serve only as a lower bound although they represent actual compile-times due to the fact that these numbers were gathered only for headers continuously *#include*-ed at the head of the source files. Since C and C++ allow header files to be *#include*-ed anywhere in the source file, an arbitrary sequence of header files culled by separating the interrupting source lines, may not compile successfully due to dependencies on the lexically preceding source. This is addressed in greater detail in the next section.

Table 3 clearly demonstrates that a large part of the compile times of these applications is spent in processing the *#include* header files. This argues for a compilation technique that minimizes the overhead of processing headers.

Table 1

APPLICATION	PREPROCESSED LINES P_l = wc -l *.i	SOURCE LINES S_l = wc -l *.C	HEADER LINES $H_l = P_l - S_l$
Hw.c	341	6	335
Hw.C	679	6	673

Table 2

APPLICATION	PREPROCESSED LINES P_l = wc -l *.i	SOURCE LINES S_l = wc -l *.C	HEADER LINES $H_l = P_l - S_l$	% HEADER LINES
Perl	69958	23,214	46744	66
Class Library	177164	8372	168792	95
Ray Tracer	425717	18323	407394	95
CAD	443358	9114	434244	97
Web Browser	48920	1002	47918	97
Linker	1060368	36832	1023536	96
Optimizer	2979219	222367	2756852	92
Business Planner	279107	102597	176510	63
Average				87

Table 3

APPLICATION	COMPILE TIME OF PREPROCESSED LINES (SECS)	COMPILE TIME OF HEADER LINES (SECS)	% TIME SPENT COMPILING HEADERS
Perl	7.3	3.9	53
Class Library	17.2	17.1	72
Ray Tracer	44.7	38.1	85
CAD	38.7	31.3	80
Web Browser	14.6	14	95
Linker	85.8	77.6	90
Optimizer	194.2	146.5	75
Business Planner	70.5	34.6	49
Average			74

Precompiled headers (PCH) are a mechanism to cache a partially compiled version of one or more headers during a compile and then, reuse the cached version during subsequent compatible compiles. To elaborate, when the compiler is invoked on a source file *a.c* for the first time, the PCH mechanism snapshots the *#include* headers in a partially compiled state and preserves that intermediate form in a disk cache. Later, when *a.c* is edited and recompiled, the precompiled contents of the cache are simply ingested instead of recompiling the same headers again. This results in reducing a portion of compile time spent in processing the headers during the second and subsequent compiles.

There is, of course, the overhead of dumping the precompiled headers during the initial compile and the expense of ingesting them and reconstructing the compilation such that it mimics the actual reprocessing of the headers during the later compiles. In spite of this, in applications with large existing source bases that change incrementally, significant portions of the code remain dormant and are therefore prime candidates for PCH. In these cases the PCH scheme yields noticeable compile-time savings.

Several commercial compilers support some form of precompiled headers including Borland C/C++ *[5]* and IBM OS/390 C/C++ *[3]*. IBM C Set ++ *[4]* caches tokenized versions of headers while aCC's implementation caches headers in a partially compiled form that results after parsing and semantic analysis. KAI C++ *[2]* dumps and reads its raw symbol table contents while aCC serializes the symbol table and the intermediate form of the headers into the cache. Microsoft VC++ *[6]* offers automatic precompiled headers with a cache that is shared across a project and causes debug information to be emitted into every object file that uses the cache. This creates dependencies between the object files that share a cache and causes the compiler to rebuild all object files that use that cache if one changes. aCC's implementation does not share the PCH across source files. Other research directions for faster compiles include incremental compilation at a function level *[7]* and a compile server approach that retains internal data structures across compile requests *[8]*. Note that our paper is the only description of details that an implementation has to take care of.

Design Issues with Precompiled Headers

The basic idea of precompiled headers is simple: avoid reprocessing the *#include* headers each time a file is compiled by reusing a partially compiled version of the same headers from a cache. In the code sequence below, the compiler could simply cache the partially compiled contents of *a.h* and *b.h* in Example 1 into a single PCH cache for later use.

Example 1

```
// Start a.c
#include "b.h"
// Other C/C++ code ...
#include "a.h"
// Other C/C++ code ...

int main () {
// More code...
}
// End a.c
```

However, complications arise due to language and compilation system features that impose constraints on the save and reuse mechanisms. Such constraints impact the extent and contents of the region that can be precompiled and control the conditions under which the precompiled cache can be reused. The main design constraints are due to the:

- scope of the macro language in C/C++
- effect of external compile environment
- flexibility of header files

The C *[11]* and C++ *[10]* languages support a globally scoped macro preprocessor language that can penetrate any lexically succeeding header file boundary. This means that the contents of header files can be manipulated through macro symbols *external* to them. For instance, given the following contents of *a.h,*

Example 2

```
// Start a.h
#ifdef HPUX11
typedef size_t unsigned long
#else
typedef size_t unsigned int
#endif

struct List {
// More code...
};

#ifdef VER2
// More code...
#endif
// End a.h
```

and a source file *a.c*, that *#include*-s *a.h* as follows,

```
// Start a.c
#include "b.h" ←May change a.h!
#define HPUX11 ←Changes a.h!
#include "a.h"
#define VER2 ←No effect on a.h


int main () {
// More code ...
}
// End a.c
```

if the macro HPUX11 is defined or undefined in Example 1, the contents of *a.h* as visible to *a.c* is affected. Therefore, in order for *a.h* to be cached and reused correctly, either its partially compiled state should reflect the presence of the macro conditional in it, or the cached version should be sensitive to the lexically preceding macro settings in the source file that influence its contents. If not, the cached version of the header file is inapplicable in contexts with different macro settings.

Further, although macros that lexically succeed the *#include* statement do not exert any influence on its contents, C/C++ compilation systems do allow for macros to be set and unset through command-line options to the compiler. For instance, the *HPUX11* macro in *a.h* above could be set or unset through:

```
aCC -c -DHPUX11 a.c
        or
aCC -c -UHPUX11 a.c
```

Such macro settings affect the contents of all dependent header files and hence the contents of the precompiled cache. For a rigorous quantification of the incidences of macros, analysis of their usage patterns and their impact on development tools see *[1]*.

In addition to the compiler options that flag macros, options that control optimization levels and debug-information generation affect the generated code and hence impact the precompiled cache contents. For instance, in a non-debug-mode compile, the partially processed state for *a.h* may not have debug-information annotations for the *List* data-type and hence, neither will its precompiled cache. In a later compile, if the debug option is set and this cache is reused, no debug-information will be emitted for *List*. This is both unexpected and inconsistent with the current invocation of the compiler. Similar incompatibilities arise with options that control 32/64-bit code generation, exception-handling etc. So, for the correct reuse of the cache, the PCH mechanism must be aware of any compiler option settings that violate its consistency.

The above discussion shows that the PCH cache is affected by more than merely the headers' contents themselves. In fact, since macros preceding the *#include* statements can affect the contents of those header files, other lexically preceding header files can also toggle these macro values and hence alter header file contents. For instance, in Example 1, *b.h* is included before *a.h* and hence can affect the contents of *a.h* by changing the settings of macros that *a.h* depends on. This implies that if the user edits *a.c* to include *b.h* after *a.h* instead of before, the existing precompiled headers cache is rendered useless! In essence, the order of inclusion of header files important to the reuse of the header cache.

Also, many C/C++ language implementations support a variety of *pragmas* that control object layout, alignment, optimizations, inlining and code-generation. These, of course, apply to lexically following code, perhaps including header files. Furthermore, since header files in C/C++ are not required to be insular entities whose contents are complete, correct and guaranteed to compile in a stand-alone fashion, header files that compile in one context may fail in others. For instance,

Example 3

```
// Start b.c
static
#include "c.h"

extern void foo (int);
int main () {
foo (I);
}
// End b.c

// Start c.h
int I;
// End c.h
```

the code in Example 3 may be stylistically deplorable but it is perfectly legal. This implies that *#inlude*-d header files contents are highly context sensitive and impacted substantially by preceding text.

Implementation Issues with Precompiled Headers

Based on the previous discussion, in order to correctly capture the header files included in a given source file

in a partially compiled state, an implementation must identify two things. First, it must demarcate the region to precompile, called the *passive region*, and then it must identify the parameters that influence the contents of this region, called the *configuration*.

Once these two entities are identified, they can be recorded in the cache when the source file is compiled. When a recompile is initiated by the user or the build system due to say, source-file changes or compile environment changes, the PCH mechanism must check to see if the *passive region* is untouched in the source file. It must also verify the compatibility of the *configuration* that controls the contents of the *passive region*. If these two attributes are intact, the cache can be reused for a faster compile.

The *passive region* is loosely described as a set of header file includes and it's preceding code, which impacts their contents. In identifying the start of this region, recall from the previous discussion that any source statements that precede the header files impacts their contents. Given that C/C++ source files are typically laid out with a series of header file include statements at the top, the *passive region* could simply begin at the head of the source file. In finding the end of this region, recall from the previous discussion that header file include statements can occur anywhere in the source file.

This implies that the end of the *passive region* could potentially coincide with the end of the source file causing the cache to contain the whole contents of the source file. Ostensibly, such a cache would derive maximal compile-time savings from reuse. In reality, such a cache would result in no savings at all! If the source file source is edited and recompiled, a cache that encompasses the gamut of the source file is automatically rendered void since no matter what part of the source file or its headers are altered, the cache is affected and cannot be reused.

So, the key to identifying the end of the *passive region* is to recognize that the overriding objective of the PCH mechanism is to reduce compile-times not by enlarging the volume of the PCH cache but by increasing the chances of its reuse. For existing programs with a largely frozen interface, the application header files seldom change and the system header files are immutable by the user.

Given this, a *passive region* that terminates with the first non-preprocessor non-comment statement in the source file is likely to cover any introductory comments, initial macro defines and conditional compile statements abetting the header file includes, but little else. This ensures that the *passive region* is not too trivial to yield benefits from precompiling. It also ensures that the *passive region* is limited and does not overrun the implementation in the source file that is prone to change more frequently. Example 4 illustrates the *passive region* boundaries in sample code.

Example 4

```
// Start foo.c
/* ... */

        ← Start passive-region
#include <iostream.h>
#include <new.h>

#ifndef __GNU__
#define MIN(a,b) ((a<b)?a:b);
#endif

#ifdef __HPUX__
#include "hp_stack.h"
#else
#include "stack.h"
#endif
#include "foo.h"
          ← End passive-region

extern void foo (void);
int lookup () {
foo ();
// More code...
}
// End foo.c
```

A source file needs to be recompiled either because its contents changed or the contents of one or more of the directly or indirectly included headers changed or because the compile environment (e.g. command-line options, compiler version) changed. This threefold collection of parameters that impacts the contents of the *passive region* is called its *configuration* and the PCH mechanism must predicate the reuse of the header cache upon the compatibility of the *configuration* including that of the *passive region*.

The first *configuration* parameter is the contents of the source file. A change to the source-file contents could either lie within the *passive region* or outside it. Since the *passive region* begins at the top of the file, any alteration outside that region of code must be lexically beyond its scope and influence. This means that a source file change either directly touches the *passive region*, by say, adding or removing header file include statements or macro settings in it, hence violating the validity of the cache, or is extraneous to the *passive*

region and does not prevents cache reuse. By recording the *passive region* itself, in addition to saving its precompiled form, the PCH mechanism can compare it against the version in the source file as a deciding factor for cache reuse during a later compile.

The second *configuration* parameter is the contents of all the direct and indirect header file dependencies of the source file. If the user edits one or more of the application headers to augment or alter its contents and one or more of these headers are included in the *passive region,* this nullifies the contents of the header cache. Therefore, in addition to a copy of the *passive region,* the header cache must contain the time-stamps of all dependent header files for the source file. This can then be checked by the PCH mechanism during a recompile to decide if the cache is worthy of reuse.

The third *configuration* parameter is the environment of the initial compile of the source file. This includes compilation command-line options, the compiler and/or PCH version and perhaps the OS versions and the compile location. Note that some compiler options like say, the verbose option, may be benign even if flipped and the partially compiled forms of the parse trees of various compiler versions may be in fact be compatible. However, for the sake of simplicity, the PCH mechanism can safely record all these parameters during the initial compile and check their consistency during a recompile.

The process that automates the decision making for creating and reusing the PCH can be implemented as follows. When the compiler is invoked on a source-file it checks to see if the corresponding PCH exists. If not, it proceeds to create one along with a record of its *configuration* parameters. If the PCH exists but its *configuration* is mismatched with the current invocation, then the compiler pretends that the PCH is absent and generates one. If a valid PCH exists, the compiler simply absorbs it, skips past the *passive region* and continues the compile.

In aCC's implementation, when it is first invoked on a source-file, it checks for the presence of a matching header cache. If no such exists, it slips into the *create* mode, in which it prepares to dump a PCH file with its associated *configuration.* First, it assembles a *configuration-record (CR)* that encapsulates the entire compile environment of the source file and its dependencies. The *CR* contains the following,

- the source filename
- the aCC version number
- the aCC command-line with all options
- header-file dependencies
- the time-stamps of the dependencies.

The OS version is not needed since aCC's version number uniquely identifies the major OS streams. In fact, the compiler version number serves to version the PCH, ensuring that the internal structure and format of the header cache are consistent with the expectations of the compiler i.e. what is written into the cache is what is read. In addition, it also captures a representation of the source in the *passive region.*

Note that the compile directory is not deemed necessary since the chances of erroneous cache reuse with eponymous files in two distinct locations are minimal. This is because aCC's implementation stores and verifies the *passive region* and the names and time-stamps of the dependencies before the reuse anyway.

aCC starts the *passive region* at the first token in the source file past any comments and white-space characters. In its search for the end of the *passive region,* it then skips past the preprocessing directives until it reaches the first declaration in the source file aCC then pre-compiles the portion of the program within the *passive region* and dumps it into an eponymous header cache with its *CR*. It also stores a copy of the actual source code from the *passive region* in the *CR*.

In determining the end of the *passive region,* if the sentinel declaration occurs inside a conditional compile block, aCC raises the end of the *passive region* above the conditional compile block. Similarly, if aCC spots a header-file include statement inside a nested scope like a class or function, it stops the *passive region* in the file-scope prior to the start of the nested scope. This is shown in the code fragments below in Examples 5 and 6.

This is because, during subsequent recompiles of the source-file when the header cache is reused, aCC pretends that the source-file begins after the *passive region.* In this mode, called the *use* mode, aCC skips past the *passive region* to start the compile and the contents of the header cache are paged in only as needed. Tokenizing and parsing of the source-file begin immediately following the *passive region* and this requires the aCC scanner and parser to encounter legal start tokens in an outermost scope just beyond the *passive region.* In effect, the source-file without the *passive region* is expected to be a syntactically correct entity.

Example 5

```
// Start foo.c

            ← Start passive-region
#include <iostream.h>
#ifdef __HPUX__
#include "hp_stack.h"
#else
#include "stack.h"
#endif
#include "foo.h"
            ← End passive-region

#ifndef VERSION10
extern void bar ();
#endif
// End foo.c
```

Example 6

```
// Start foo.c
            ← Start passive-region

#include <iostream.h>
#include "foo.h"
            ← End passive-region

struct Tree {
#include "bar.h"
// More code...
};
// End foo.c
```

In addition, aCC supports a *pragma* that can be inserted by the user to define the end of the *passive region*. Such fine-grained control is convenient, for instance, when the user knows that a specific header file is currently in flux and therefore not amenable to precompile and wants to assert this to the compiler. In the absence of such a manual control, the automatic stop-point of the *passive region* may fall beyond the header in flux and hence negate any benefits that accrue due to the rest of the *passive region* preceding it.

Clearly, the manual stop-point must lexically precede the automatic stop-point to have effect. Of course, for reasons already discussed, the start of the *passive region* must always fall at the top of the file and hence is not open to user manipulation. Comments at the head of the source-file are not included in the *passive region* since they are semantically immaterial.

In *create mode,* once the *passive region* is identified and the *CR* assembled, the next step is to dump a partially compiled version of the headers into the cache. In aCC, partially analyzed parse-trees decorated with their symbol and type attributes are serialized to populate the precompiled header database during the *create mode.*

In *use mode*, once the configuration parameters mentioned above have been validated, the linearized parse-trees are read back from the disk cache on demand with only the cache preamble read in at start of the compile. As the compile progresses in *use mode,* if a required symbol or type is missing, the lookup fails and thereby triggers the PCH load mechanism for that symbol or type. Then, the symbol and its attributes are read in from the cache to fill in the in-core symbol table.

Results and Correctness

Table 4 presents the results of the automatic precompiled header implementation in aCC for complete end-to-end builds. The Web Browser, CAD and Business Planner are application subsets while the rest are complete applications. This implementation is entirely Makefile transparent and can be enabled and disabled through options or an environment variable.

As shown below, the average compile-time speed-up for these applications is over 2x. Perl has the least compile-time gains and the Web-Browser subset, the most. This matches the expectations set by Tables 2 and 3. Perl has the smallest ratio of header contents to source-file contents and the least time spent in compiling its headers while the Web Browser subset has a substantial portion of its code in its headers and it consumes a major fraction of its compile-time. Note that the cache is managed by aCC and not Make; it is therefore reused based on its validity or recreated by aCC. If the user chooses to explicitly delete the whole cache before the build, there are obviously no compile-time gains.

Table 5 juxtaposes the compile-time gains due to precompiled headers against the cost of the initially warming the cache via the *Use Factor.* The *Use Factor* is calculated using the equation:

$$nt_1 = t_2 + (n-1)t_3$$

where t_1 is the compile-time without using pre-compiled headers, t_2 is the time to create the header cache and t_3 is the time to compile using the header cache. The *Use Factor (n)* then estimates the minimum number of times a file must be recompiled using its PCH to recover the cost of creating the cache.

Table 4

APPLICATION	COMPILE TIME OF PRE-PROCESSED SOURCE (t_1) (SECS)	COMPILE TIME FOR PCH USE MODE (t_2) (SECS)	SPEED-UP (t_1 / t_2)	% SPEED-UP
Perl	7.3	5.7	1.2	21
Class Library	17.2	7.4	2.3	56
Ray Tracer	44.7	19.1	2.3	57
CAD	38.7	18	2.1	53
Web Browser	14.6	3	4.8	79
Linker	85.8	21.1	4.0	75
Optimizer	194.2	76.9	2.5	60
Business Planner	70.5	43.8	1.6	37
Average			2.6	54

Table 5

APPLICATION	COMPILE TIME OF PRE-PROCESSED SOURCE (t_1) (SECS)	COMPILE TIME FOR PCH CREATE MODE (t_2) (SECS)	COMPILE TIME FOR PCH USE MODE (t_3) (SECS)	USE FACTOR (n)
Perl	7.3	11.3	5.7	3.5
Class Library	17.2	22.8	7.4	1.5
Ray Tracer	44.7	66.7	19.1	1.8
CAD	38.7	56.1	18	1.8
Web Browser	14.6	16.2	3	1.1
Linker	85.8	124.9	21.1	1.6
Optimizer	194.2	263.7	76.9	1.5
Business Planner	70.5	78.5	43.8	1.2
Average				1.5

Table 6

APPLICATION	SIZE OF OBJECT FILES (KB)	SIZE OF HEADER CACHE (KB)	RATIO OF CACHE SIZE TO OBJECTS SIZE
Perl	508.1	4699.9	9
Class Library	644.2	10324.8	16
Ray Tracer	1338.8	30714.2	22
CAD	1440.0	21490.6	14
Web Browser	392.4	3424.9	8
Linker	2346.9	61472.5	26
Optimizer	9644.9	107103.5	11
Business Planner	4855.6	12900.8	2
Average			13.5

Table 6 displays the size of the object files versus the size of the precompiled header cache for each of these applications. Since the cache occupies a significant amount of disk space, aCC provides an option to redirect it to a different disk or directory other than the default source file directory. This option can also be used to maintain multiple precompiled header caches, each for a different build target like say, debug build or optimized build.

In compiling these real applications, some correctness related issues surfaced due to:

- synthesized header files
- *__DATE__* and *__TIME__* macros
- revision control systems

One application recreated a set header files afresh for each build. This dynamic generation of header files at build-time effectively thwarted aCC's precompiled header mechanism. The generated header file time-stamps were always more recent than those recorded in the cache *CR* and hence would negate the cache on each build! This problem caused aCC to perpetually stagnate in the *create mode,* thus penalizing the application compile time further through the cost of creating the cache.

On examining this closely, it emerged that in actuality the generated header files were materially unaltered each time. That is, the contents of these header files did not change each time they were synthesized although their time-stamps did. Hence, it was clear that the header cache was in essence reusable since its contents were valid. In order to establish that, the cache *CR* was augmented to contain the checksum of the header dependency in addition to its name and time-stamp. aCC would now compute the checksum based on the contents of each header and include it in the *CR* along its names and time-stamps.

This technique also addressed the problem of time-stamp comparisons in the presence of multiple machines with possibly non-synchronous clocks. Further, in environments that upgrade to new system releases, the system and library headers' time-stamps are changed even when there are no material changes to them. Capturing the checksum in addition to the dependency time-stamp handles this situation as well.

Another application used the *__DATE__* and *__TIME__* macros in header files that were deemed potential hazards in using precompiled headers. While the C/C++ languages promise that the *__DATE__* and *__TIME__* macros will reflect the date and time during the translation of the source file, they do not dictate when that particular instance of translation must terminate. To elaborate, a compiler could start translating a source file with an occurrence of the *__TIME__* macro at 5.00 AM while compiling another such occurrence in the same file at 5.05 AM. Similarly, precompiled headers with occurrences of such macros could reflect the date and time during their translation, which may be different from when the rest of the source file is compiled or recompiled.

This behavior should be noted and clearly understood that when previously compiled components are integrated into a more recent set of compiles there may exist discrepancies in temporally sensitive code. One way to deflect the problem is to have the compiler abort the creation of the header cache for the current source file if it encounters these macros. This will result in a normal and correct compile but without the benefits of PCH. Another alternative is to accord special treatment to these macros. Typically, in aCC, the definitions of macros are stored in the header cache verbatim but their uses in the headers being precompiled are recorded after the macro is substituted. These two macros could be saved in their original form and hence recalled and replaced when the actual recompile happens, thus reflecting a more current date and time. aCC simply acknowledges this but does not yet implement this alternative in its precompiled header mechanism.

Some precompiled header implementations are based on the premise that source files in an application share a common set of headers that incur repeated processing in the course of building the application. Therefore, by factoring out the commonality into a precompiled form, the compiler saves on redundant processing between source files in addition to repeated processing within them. This scheme may also result in smaller header cache sizes but may require the user to reorganize the sources to *#include* a common set of headers in a specific order that can then be precompiled into a shared database. However, aCC's scheme does not share precompiled headers between source files and this quality in turn, enhances its usability in common application build environments.

User applications often rely on source code control and versioning software to select, build and maintain correct versions of their source files. One such commonly used system is *Clearcase [9]. Clearcase* maintains clear correspondences between source file elements, their dependencies and the object files that derive from them. By creating a header cache that is common to several source files, a new constraint is introduced on all the object files that are derived from each of those sources.

For example, say that the source file *a.c* containing header files *a.h* and *foo.h* produces a header cache that is shared by *b.c* since it also included the same headers. *Clearcase* then records this cache as a dependency for both *a.o* and *b.o.* Later, say *a.c* is edited to remove *a.h* and recompiled, then the cache is deemed invalid for reuse with *a.c* and the compiler enters the *create mode.* It rewrites the cache and creates a new object file, *a.o.* Since that cache has been marked as a dependency for *b.o,* it triggers a rebuild of *b.c.* This renders the same cache unusable for *b.c* since it now contains different headers and recreates it! This cross dependency can confuse *Clearcase* and lead to a deadly cycle of rebuilds that immobilizes productivity.

aCC's implementation of the header caching scheme attaches a unique PCH to every source file, thus eliminating any dependency between the cache and

multiple object files. Note that the overriding goal in this of PCH is to favor the correctness of results over maximizing the reuse of the PCH. This implementation thus trades-off any savings in disk space that may accrue from sharing the PCH cache across many source files for improved usability with real application development environments. This may be considered a cost-effective trade-off since the effects of increased disk consumption may be mitigated to some extent through cache compression.

Not including the compile directory in the *CR* also allows various users compiling the same file from different directories to reuse the object file and cache. This does not compromise the correctness of this scheme since over and above the *CR*, the *passive region* must also match.

Other applications mimic source-code control systems in a limited fashion through *view-pathing*. *View-pathing* is a feature that delineates a set of paths for the master-copies of certain header files while reserving another set for holding the user-modified versions. Through the flip of a switch the user can select the preferred version over the original version.

For example, let *a.c* include *a.h* and *a.h* reside in the directory *sysinc* during an initial compile. Let the user now copy *a.h* to another directory, say *userinc*, and edit it. If the user now recompiles the file, the *passive region*, the *CR* and the originally recorded set of dependencies are proven valid and the cache is reused. However, the desired alternate header file is not selected!

```
#Create cache
aCC -c -Isysinc -I- -Iuserinc +hdr_cache a.c

#User copies a.h and modifies it. Use cache!
aCC -c -Isysinc -I- -Iuserinc +hdr_cache a.c
```

aCC's implementation of precompiled headers is insensitive to *view-pathing* and hence is blind to the additions of header file to alternate locations in the list of *-I* directories. One possible remedy may be to augment the *CR* with the *-I* directories' contents and detect any changes in these prior to using the header cache.

Conclusion

An automatic header caching mechanism has been implemented in aCC and is available in its currently shipping versions. The full implementation including the load-dump mechanism (not discussed in this paper) consists of about 14,000 lines of commented C++ code. This exercise helped identify several key design and implementation issues due to the language definition and existing compile environments in providing a usable and fully automatic precompiled header mechanism. This paper discusses many of these issues, describes aCC's implementation choices and trade-offs and suggests remedies to open issues. The advantages of this scheme are:

- Ease of use. It requires no manual intervention. Source files need not be tailored to contain *#include*-s in any specific order. Header files need not be altered to have include guards around them.
- It is transparent to the Make process and friendly to revision control systems

The biggest limitation is that the header cache consumes a significant amount of disk space. Finally, this paper presents the results of using aCC's PCH mechanism on a set of applications. It improves compile-times by over 50%.

References

[1] Michael Ernst, Greg Badros, David Notkin. *An Empirical Analysis of C Preprocessor Use.* Technical Report UW-CSE-97-04-06, Department of Computer Science and Engineering, University of Washington, Seattle.

[2] *KAI C++ ™ User's Guide, Precompiled Headers* http://www.kai.com/C_plus_plus/v3.4/doc/UserGuide/precompiled-headers.html

[3] *IBM OS/390 V2R6.0 C/C++ User's Guide, Chapter 10 Using Precompiled Headers.* http://www.s390.ibm.com:80/bookmgr-cgi/bookmgr.cmd/BOOKS/CBCOUG03

[4] *IBM C Set ++ for AIX.* http://www-4.ibm.com/software/ad/caix/

[5] C. Horstmann. *An in-depth look as Borland C++.* C++ Report, 3(10), pp. 17-20, Nov-Dec 1991.

[6] *Microsoft VC++ Language Help, Automatic Use of Precompiled Headers.*

[7] Fyfe, Soleimanipour, Tatkar. *Compiling from Saved State: Fast Incremental Compilation with Traditional; Unix Compilers.* Usenix, Winter 1991.

[8] T Onodera. *Reducing Compilation time by a Compilation Server.* Software Practice and Experience. Vol 23(5), May 1993.

[9] Rational Software Corporation, *Clearcase* http://www.rational.com/products/clearcase/

[10] *ANSI/ISO C++ Final Draft International Standard*

[11] *ANSI C Standard*, ANSI ISO/IEC 9899:1990

[12] *aCC : The HP ANSI C++ Compiler.* http://www.hp.com/go/hpc++

C++ Exception Handling for IA-64

Christophe de Dinechin
Hewlett-Packard IA-64 Foundation Lab
ddd@cup.hp.com

Abstract

The C++ programming language offers a feature known as exception handling, which is used, for instance, to report error conditions. This technique can result in more robust software. On the other hand, it generally has a highly negative performance impact, even when exceptions are not actually thrown. This impact is especially important on an architecture such as the HP/Intel IA-64 processor, which is very sensitive to compiler optimizations. Hewlett-Packard implemented exception handling for IA-64 in a way that leaves the door open for optimizations, even in the presence of exceptions.

1. Overview of C++ Exception Handling

Most software has to deal with exceptional conditions, such as insufficient resources, missing file or invalid user input. In C, such a condition is typically reported using special return codes from functions. For instance, the ubiquitous `malloc` function indicates an out-of-memory situation by returning a NULL pointer. Typical C code would test this situation as follows:

```
void *ptr = malloc(1000000);
if (ptr == NULL)
   fprintf(stderr, "Sorry, out of memory\n");
```

C++ exceptions are a better way to report such a condition. A C++ function that detects an exceptional situation can *throw an exception*, which can be caught by any of the calling functions using an exception handler. For instance, the previous code could be written in a C++ program as follows (the error test is in bold):

```
struct OutOfMemory {};
struct Resource {
   Resource();    // Ctor allocates resource
   ~Resource();   // Dtor frees resource
};

int foo(int size) {
   void *ptr = malloc(size);
   if (ptr == 0)
      throw OutOfMemory();
   /* Do something else */
}

int bar(int elements) {
   Resource object;
   int result = foo(2 * elements);
   /* Do something else */
}

int main() {
   int i;
   try {
      for (i = 0; i < 100; i++)
         bar(i);
   } catch (OutOfMemory) {
      /* Report out-of-memory condition*/
      cerr << "Out of memory for i="
         << i << endl;
   } catch (...) {
      /* Report other problems. */
   }
}
```

If the anomalous situation is detected (in this case, `malloc()` returning zero), the function can report it by throwing an exception. Note that this would not even be necessary had the memory allocation been done the C++ way, since the C++ allocation operators normally report an out-of-memory condition by throwing a standard exception (`std::bad_alloc`). Compared to the C error reporting method, the benefits are multiple:

- The exceptional situation is identified by a specific entity, an exception, rather than by a special return code.
- There is no need for intermediate functions, such as `bar`, to do anything to deal with the exceptions.
- In particular, objects with destructors such as `object` are properly destroyed when the block containing them is exited, whether normally or because of an exception. This makes resource management safer.
- The exception handling code (in `main`) is easily identified as such, and separate from normal processing. A `catch` block catching the exception type `OutOfMemory` is called an exception handler for `OutOfMemory` exceptions.

Throwing an exception involves *unwinding* the call stack until an exception handler is found. This process is made more complex in the presence of C++ automatic

objects, since these objects may have destructors. In that case, destructors have to be called as the stack is being unwound.

2. Performance Impact of Various Solutions

Since exceptions occur infrequently, the performance of exception handling code is normally not critical. In addition, developers can easly control how their application uses exceptions, and avoid exceptions in performance-critical code.

On the other hand, implementations of exception handling generally have a negative performance impact on the code that may throw an exception (the code inside a `try` block), whether this code actually ever throws an exception or not. Ideally, the code inside the try block should not be different than the same code outside a try block. In practice, however, the presence of a try block, or even the presence of an "exceptions are enabled" option in general slows down the code and increases its size. The reasons are multiple and complex. We try to address some of them below.

The performance of an exception-handling solution is therefore measured by its impact on the non-exceptional code when no exception is thrown; it should try to minimize the degradation of code speed and size for this "normal" code.

2.1 Portable Exception Handling with setjmp

The first implementations of C++ exception handling used a mechanism based on the standard C `setjmp` and `longjmp` functions. The `setmp` function saves an execution context in a `jmp_buf` structure. The `lonjmp` function can later be used to perform a "non-local goto", transferring control to the place where `setjmp` was originally called, as long as the function containing the `setjmp` never returned.

In "portable" exception handling, a try block is replaced with a `setjmp` call, and throwing an exception is replaced by a `longjmp`. A linked list of `jmp_buf` buffers will represent the dynamic list of enclosing try blocks. This same technique had been used routinely in C and C++ to simulate exceptions before exceptions became available as a standard language feature.

The major difficulty with this approach is to correctly destroy automatic objects (such as the `Resource` object in our example). This is typically solved by creating a linked list of the objects to be destroyed as you create them. This approach is relatively simple, and it works with a C++ compiler that generates C code, such as the original Cfront from AT&T — this is the reason it is called "portable". This scheme has been used quite widely, in particular by the Cfront-based C++ compiler from Hewlett-Packard [1].

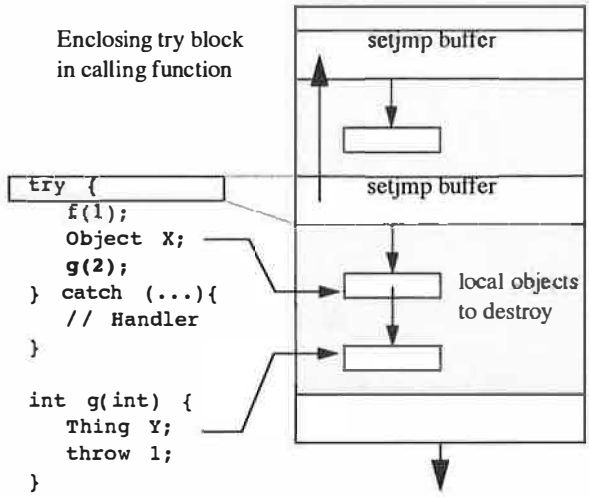


Figure 1. Setjmp based exception handling

On the other hand, the performance drawbacks are significant.

- The `setjmp` function must be called at the beginning of every try block, and the list of `jmp_buf` must be maintained.
- A linked list of objects on the stack must be maintained at all times, and kept in a consistent state with respect to the list of `jmp_buf`.
- All variables that are stored in registers and that are declared outside the try block have to be restored to their initial value when `longjmp` is invoked[1]. For instance, the value of `i` in the `catch` block in `main` must be the same value as when `bar` was called. This can be achieved either by spilling all variables to memory before calling `setjmp`, or by having `setjmp` itself save all registers. Both options are expensive on architectures with large register files such as RISC processors.

This impact exists even if no exception is ever thrown, since the calls to `setjmp` and the management of the

1. Typically, `setjmp` will not save all registers in the `jmp_buf` it is given as an argument. This is why the documentation for these routines generally states something like: "*Upon the return from a setjmp() call caused by a longjmp(), the values of any non-static local variables belonging to the routine from which setjmp() was called are undefined. Code which depends on such values is not guaranteed to be portable.*" (from the HP-UX 10.20 man page for `setjmp`).

object stack have to be done each time a try block is entered or exited.

2.2 Table-Driven Exception Handling

Another implementation of C++ exception handling uses tables generated by the compiler along with the machine code. When an exception is thrown, the C++ runtime library uses the tables to perform the appropriate actions. Conceptually, this process works as follows:

- A first table is used to map the value of the program counter (PC) at the point where the exception is thrown to an action table.
- The action table is used to perform the various operations required for exception processing, such as invoking the destructors, adjusting the stack, or matching the exception type to the address of an exception handler. For example, there will be an action kind to indicate "call the destructor for object on the stack at stack offset N," which will be used to invoke the destructor of the `Resource` object.
- Once an exception handler (a `catch` block corresponding to the type of the exception being thrown) is found, a new PC value is computed from the tables that corresponds to this handler, and control is transferred to the handler.

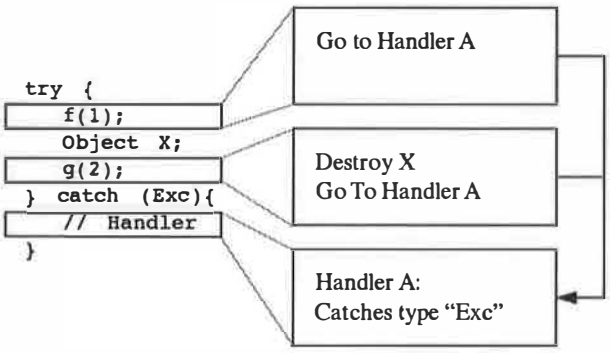


Figure 2: Table Based Exception Handling

This approach is significantly more efficient than the previous one. There is no longer the systematic cost of a `setjmp` function call for every `try` block. Similarly, the cost of maintaining linked lists even when exceptions are not thrown is also eliminated. Therefore, many C++ compilers switched to a table-driven exception-handling mechanism. The Hewlett-Packard aC++ compiler for PA-RISC uses this technique.

On the other hand, there are still negative effects from a performance point of view:

- The runtime needs to be able to restore all variables that are declared outside the try block to their correct value before entering a catch block (for instance `i` in the `catch` block of `main` in the example above.) The impact of this on performance is quite subtle and has multiple aspects, which are discussed below.
- All objects that have destructors must have their address stored in a table. Therefore, they must reside in memory, and their address is implicitly exposed.
- All automatic objects that have their address exposed have to be committed to memory before any call. In practice, this is not often a significant constraint, since a C++ object's address is exposed through the `this` pointer after any member function call (including the constructor.) On the other hand, this may impact the most performance-critical objects, whose member functions are all inlined. These objects could otherwise be promoted to registers.
- The tables themselves have to encode a lot of possible actions, including call to destructors. Therefore, they use a significant amount of space.
- Since tables refer to code, reorganizing the code implies reorganizing the tables accordingly. While this does not preclude optimizations on a theoretical ground, it practically disqualifies any existing optimizer that does not specifically know about C++ and the exception-handling tables.

2.3 Extension of Variable Lifetime

The following code illustrates one problem related to preserving the value of local variables in the presence of exception handling:

```
void f() {
   int x = 0;
   x = f1(x);
   f2(x);
}
```

A smart compiler can discover that the only use of the initial value of `x` is for calling `f1`, at which point it is known to the compiler that the value is zero. Then, `x` gets immediately overwritten with a new unknown value. Therefore, the compiler can legally rewrite the code as follows:

```
void f() {
   int x = f1(0);
   f2(x);
}
```

However, if the above code were to be placed in a try block, this transformation would no longer be valid. For instance, the value of `x` could be used in the catch block:

```
void f() {
   int x = 0;
   try {
      x = f1(x);
      f2(x);
```

```
    } catch (...) {
       cout << "The value of x is " << x;
    }
}
```

This phenomenon extends the lifetime of a variable, and therefore puts additional pressure on the register allocator. It also makes the control flow much more complex, by creating additional potential control-flow arcs between any call and each of the catch clauses. As a result, register usage will tend to be much more conservative within a try block than outside of it. On the other hand, these effects occurs only in the presence of a try block: destructors, for instance, cannot access local variables whose addresses have not been exposed.

2.4 Register Selection Constraints

Another slightly different problem can be shown in the following code:

```
int f(int x) {
   x = f1(x);
   return f2(x);
}
```

A smart compiler can notice that the initial value of `x` becomes "dead" right after the call to `f1`, since the result overwrites the previous value of `x`. The same thing happens with the second value of x, which lives only until the call to `f2`. So the compiler can rewrite the code as follows:

```
int f(int x) {
   int x2 = f1(x);     // and discard x
   int x3 = f2(x2);    // and discard x2
   return x3;
}
```

This alternative leaves much more freedom in terms of register allocation, since now different registers (or memory locations) can be allocated for `x`, `x2` and `x3`. In particular, this means that the value of the register used needs not be preserved accross the function calls. But this freedom does not exist if the above code is enclosed within a try block. In that case, the catch clause may access variable `x`, and therefore all `x` values have to live in the same register. Again, this problem occurs only in the presence of a try block.

2.5 Control-Flow Complexity

In the presence of a try block, the control flow becomes much more complex, since an implicit "goto" exists between any function call or `throw` statement in the try block and each `catch` block. Another implicit "goto" exists between the end of each catch block and the end of the function. This can impact optimizations on the following code:

```
for (i = 0; i < 1000; i++)
   x = f(i) * 3 + 1;
```

Outside of a try block, the code in question has a rather well known behavior, so if `x` is not address exposed and can therefore not be visible inside `f`, the compiler can predict that the value of x on exit from the loop will be the value computed at the last iteration. In other words, it can postpone the multiplication and addition until after the loop. In real code, the computation would probably be implicit, for instance taking the address of a struct element (an addition), or of an array element (multiplication by the element size).

Optimizing away the computation can't be done if there is a try block surrounding the code, since in that case any of the `catch` blocks can read the value of `x`.

2.6 Memory Access Order

Memory accesses are more strictly ordered in the presence of exceptions. This effect is quite significant, because it occurs even without the presence of a `try` block. Consider the following code:

```
struct Object { float x, y; ~Object(); };
Object object;
for (int i = 1; i < 1000; i++) {
   object.x += f(i);
   object.y += g(i);
}
```

In this code, the compiler can identify that for a normal iteration of the loop, memory accesses can be avoided, and replace the loop code with something like:

```
register float tmp_x = object.x;
register float tmp_y = object.y;
for (int i = 0; i < 1000; i++) {
   tmp_x += f(i);
   tmp_y += g(i);
}
object.x = tmp_x;
object.y = tmp_y;
```

Of course, if `f` or `g` can throw exceptions, then the destructor has to see the correct value of the object, and the invocation of the destructor can occur at any time. So the compiler must generate code that writes the value of the object to memory in the original source order with respect to function calls.

In practice, this last effect and its variants tends to be the most significant, since it affects memory accesses, which are expensive on today's microprocessors, and it occurs whenever exceptions are enabled, regardless of whether there are exception-related constructs in the code.

3. IA-64 Exception Handling

The various problems listed previously can be classified in one of the two following categories:

- Cost of saving and restoring registers
- Ordering constraints due to additional arcs in the control graph

The first problem is addressed in a rather original way by a feature of the IA-64 architecture called the "*Register Stack Engine*" (RSE) [3]. The RSE defines a standard way to save and restore registers when entering and exiting functions which is not directly under program control. As a result there is no real need to explicitly save registers, yet there is a way for the runtime to restore them to their original value.

The second problem is addressed in our implementation of C++ exception handling by allowing the non-exceptional path to be optimized, as long as *compensation code* is placed along the exceptional paths to restore program state before executing the exception handlers to what it would have been if the optimization had not taken place. The place where such compensation code is added is called a *landing pad*, and serves as an alternate return path for each call.

3.1 Register Stack Engine and Unwind Table

The IA-64 architecture features numerous registers [2]. The integer register set is partitioned into 32 fixed registers and 96 "*stacked*" registers. The stacked registers are automatically saved on a special stack, using free cycles in the load-store unit whenever possible.

Registers are typically not stored to memory immediately on function entry. Instead, stacked registers are renamed so that the first stacked register for the current procedure is always called `r32`. Dirty (non-saved) registers are pushed on the stack when calls are made, while previously saved registers are popped from the stack and restored when calls return. The processor tries to save as many registers for future calls and restore as many registers for future returns as free memory bandwidth and free registers allow. This technique maximizes the chances that a function call or function return can be performed without memory accesses to save or restore registers.

The Register Stack Engine which performs these operations is itself a very complex topic that would require an article in its own right [3]. For C++ exception handling, however, the key feature of the RSE is the way it transparently saves and restores stacked registers "in the background", and does so at locations on the stack that the runtime can compute.

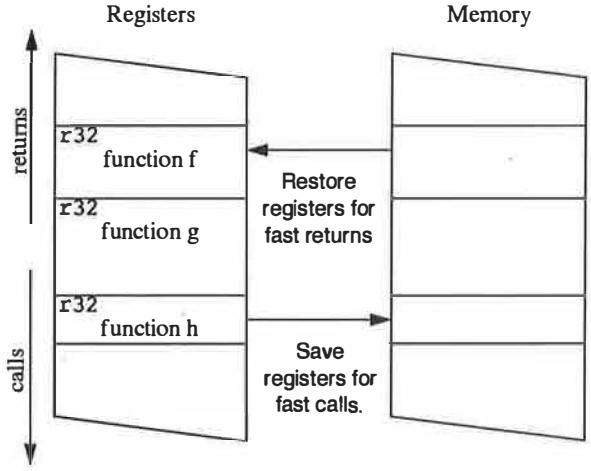


Figure 3: IA-64 Register Stack Engine

When an exception is thrown, the runtime forces the RSE to flush all stacked registers onto the stack. It can then manipulate them as needed, and later let the RSE restore them as the stack is unwound.

Only stacked registers are saved this way. The IA-64 runtime architecture [4] indicates that non-stacked registers are saved in stacked registers. Floating-point registers are saved using more traditional mechanism. The information indicating where each particular register is being saved is stored in separate tables, called *unwind tables* [5].

Together, the unwind tables, stack unwinding routines [6] and the RSE help restore register values to the exact same state they were in any given function, without the runtime cost of saving them "manually" in each function.

3.2 Exception Handling Tables

Restoring registers to their previous state is necessary, but not sufficient for throwing a C++ exception. The C++ runtime also needs to call the destructors, to find the appropriate exception handler, and to transfer control to this exception handler.

The information required to do this is found in *exception handling tables*. These tables are C++ specific. They contain information to map call sites to the landing pads. Each landing pad will process any exception thrown from the corresponding call site, and serve as an alternate return point for this call. The table also contains information regarding which exceptions the landing pad can process and catch, and records exception specifications if any.

The reason the table maps call sites is that our C++ implementation only throws from a call site. The `throw` keyword itself is implemented by calling a runtime routine. Errors such as division by zero or invalid memory accesses do not need to throw exceptions in C++ [7] (they are

"undefined behavior"). Practically all implementations use alternative mechanisms such as Unix signals[1].

Therefore, from a machine language point of view, the only places that can throw are call instructions. If the program counter is within the range of a given subroutine but no call site matches, the runtime will call the `terminate()` function, as specified in C++.

An interesting implication: this mechanism does not allow C++ exceptions to be thrown out of a Unix signal handler, something that the ISO C++ Standard specifically discourages [7] (Clause 18.7, paragraph 5, restricts signal handlers to what can also be written in plain C). For instance, if a memory access instruction causes a signal, and if the signal handler throws an exception, the program counter of the function containing the memory access will not be on a call instruction, and `terminate()` will be called.

3.3 Landing Pads

The runtime transfers control to a landing pad whenever an exception is thrown from a given call site. The landing pad will contain code in the following order:

- *Compensation code*, restoring program state to what it would be if optimizations had not been done in the main control flow.
- *Destructor invocation* to destroy any local object that needs to be destroyed.
- *Exception switch* to select which catch handler, if any, to jump to. An appropriate switch value is computed by the runtime from the C++ exceptions table, and placed in a temporary register.
- A *landing pad exit*, which either returns to a catch block, and from there to the main control flow, or resumes unwinding if no appropriate exception handler is found in this subroutine.

The same mechanism can deal with all kind of destructors (inlined or not, array destructors, ...), which all had to be special table entries in a table-driven exception handling runtime. A catch-all exception handler (`catch(...)`) is simply a default exit in the exception switch.

Together, the landing pads form a funnel where the compensation code can be somewhat different for each call site, while destructor code is shared for sections of code between declarations, and the exception switch and landing pad exit are shared for all code within the same try block.

1. One notable exception is the Win32 platform, where system exceptions and C++ exceptions interact.

3.4 Compensation Code

It is relatively easy for the compiler to generate compensation code for any of the operations listed in previous sections:

- If the variable is allocated to a different register in different sections of code, the landing pad can simply copy that register to the target register which represents the variable in the exception handler.
- A value known to be constant which has been replaced with the constant value can be loaded into the appropriate register by the landing pad, for use by the user code in the exception handler.
- Pending memory operations that have been delayed in the main flow of control can simply be executed in the landing pad should an exception be thrown.

Compensation code therefore allows the compiler to utilize any of the optimizations that were prevented by simpler table-driven techniques.

Since the IA-64 architecture is very sensitive to optimizations, the ability to insert compensation code alone is a compelling reason for selecting a landing pad based approach. On other architectures, the benefit of landing pads may not be high enough to compensate for the code size penalty compared to other table-driven techniques.

3.5 Placing Landing Pad in "Cold" Code

Landing pad code is not used except when an exception is thrown. If the landing pad code is simply placed at the end of the code for each function, the useful code becomes interspersed with blocks of little-used code. This can affect paging and caching performance, since the exception handler code will occupy space in the various memory caches and in the virtual memory active set.

For this reason, landing pad code can be placed in a different code section. This code can be placed at link time arbitrarily far from normal "hot" code. The hot code can then be kept contiguous and makes better use of the cache and virtual memory pages.

Even when it does not actually use cache lines or virtual memory active pages, landing pad wastes space on disk when it is not used. Since it is in general infrequently used, landing pad code can therefore be space optimized rather than speed optimized.

3.6 Compressing Tables

The overhead of exception handling includes the exception handling tables. These tables are not used except when an exception is thrown. Just like landing pad code, they can be placed arbitrarily far from the code so as to

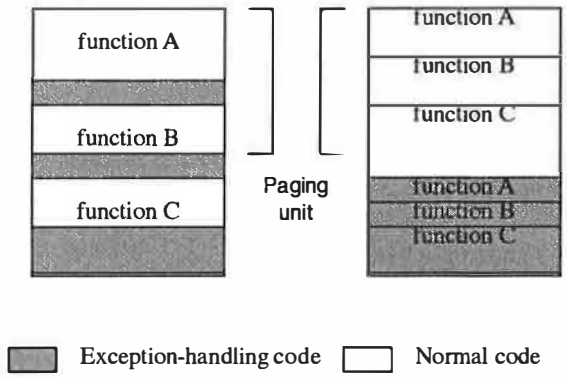


Figure 4: Separating Hot and Cold Code

minimize impact on caching and virtual memory. On the other hand, they still use valuable disk space in the executable image.

The exception handling tables in the Hewlett-Packard aC++ compiler use a compression scheme known as "LEB128". This encoding uses less space for small values: 1 byte for any value less than 128, 2 bytes for any value less than 16384 and so on. So the tables contain relative offsets that are often small. For instance, the first call site address is encoded as the number of 16-byte instruction bundles from the start of the function, and later call sites are relative to the previous call site in the table. This helps keep the offsets small enough to fit in one or (rarely) two bytes.

Compressing the tables has a slight negative effect when an exception is actually thrown, since the table contents need to be decoded. In that infrequently executed case, we traded speed for space.

3.7 Known Functions That Can't Throw

When exceptions are enabled, there is a landing pad and table space overhead for each call site. This overhead can be avoided for specific functions that are known not to throw. These functions typically include:

- C++ runtime library functions called implicitly by the compiler.
- Functions of the C library, since they are known not to throw exceptions.
- Functions that are marked as not throwing exceptions through an empty exception specification. Exception specifications are verified inside the function, not at the call site.

If such a call ever throws, the C++ runtime will call `terminate()`.

3.8 Remaining Negative Effects

Even with landing pads, exception handling still has a cost. The space overhead of enabling exception handling includes the code for the landing pads, exception switches, destructor calls and catch handlers, as well as the space for all the exception handling tables. This remains significant in terms of memory usage, even though the performance impact of this additional memory can be kept low by carefully segregating hot and cold memory.

However, performance itself can remain affected by a variety of factors:

- Additional control flow arcs between the main code call sites and the various exception handlers and destructor calls make the control flow graph much more complex.
- One result is to prevent some otherwise valid code motion, when the code motion cannot be correctly compensated for in the landing pad, or when the cost of compensating would be too high.
- Another effect is to effectively lower the amount of optimization that can be done on a given piece of code in a given amount of time. Since compilers also have a compile-time performance constraint, they may "bail out" if optimization would take too long. This will happen earlier in the presence of exception handling.
- Another instance of resource limitation occurs on optimizations that copy or duplicate code, such as inlining. These optimizations typically have a "budget", and this budget gets exhausted much more rapidly in the presence of exceptions, since duplicating the main body of code generally means duplicating the exception-handling data and code as well.
- A final problem evoked above remains unsolved: any optimizer that performs on the code has to know about exception handling tables and how to reorder them. With limited engineering resources, some specific optimizations may purposedly be disabled in the presence of exceptions.

4. Results

The following tables records timing and size measurements done on various benchmarks. These have been run on a performance simulator for the next generation IA-64 processor, and remain to be validated on real hardware. For comparison purposes, similar meaurements have been done on current generation PA-RISC processors. Only relative results are shown, since absolute SPEC results for IA-64 have not been published yet.

The values measure the performance penalty when enabling exception handling. For speed, the penalty is the additional number of cycles in the simulator. For memory, it is the additional size of text and initialized data, as reported by the `size` command.

These benchmarks contain a mix of C and C++ application code, but they often do not rely very much on C++ local objects or exception handling. Therefore, they represent a worst case but not very uncommon scenario where exception handling is not used and therefore you don't want to pay for it. Some of the benchmarks were originally written in C and have been modified to be compilable with a C++ compiler. Table 1 records size and speed penalties. In general, measurements were taken at the maximum optimization level, except for the last two rows where the optimization level was +O1.

Table 1. Speed and Size Penalty

	IA speed penalty	IA size penalty	PA speed penalty	PA size penalty
099.go	3.73%	-9.58%	15.6%	0.21%
129.compress	-5.66%	<0.01%	2.00%	< 0.01%
130.li	-6.56%	-14.19%	11.15%	1.24%
132.ijpeg	-0.21%	-0.48%	-0.49%	0.06%
134.perl	-1.43%	-18.49%	0.81%	1.02%
147.vortex	N/A	-8.08%	0.66%	-0.04%
Raytracer (+O1)	-1.6%	16.8%	0.3%	6.13%
C++ Library (+O1)	N/A	0.2%	N/A	0.2%

Suprisingly, in some benchmarks, enabling exception handling actually yields better performance. This has to be taken with a grain of salt. At this point, it is quite difficult to do accurate IA-64 measurements, whether on real hardware or on a simulator. For instance, simulator results are sampled, and the sampling noise alone can account for a few percents of variation either way. Similarly, optimizer "luck" in scheduling instructions can also introduce unpredictable variations. Therefore, the "noise level" of these measurements is quite high. You should not expect code to become faster because of exception handling.

The size aspect is even more surprising, as shown on Table 2 below.

Table 2. Size penalty for various optimizations

	+O1	+O2	+O3
099.go	9.41%	9.07%	-9.58%
124.m88ksim	18.83%	13.02%	-11.30%
129.compress	0.01%	<0.01%	<0.01%
130.li	35.73%	24.87%	-14.19%
132.ijpeg	0.72%	0.42%	-0.48%
134.perl	28.80%	28.57%	-18.49%
147.vortex	37.48%	30.57%	-8.08%

Processing exceptions requires additional code. The effect above on the IA-64 compiler shows only for the maximum optimization level (+O3), and may actually indicate a problem with the tested compiler. At lower optimization levels, the PA-RISC compiler consistently produces executables of the same size with or without +noeh. The IA-64 compiler produces executables that are significantly larger with exception handling enabled, which is what one would expect.

5. Analysis

Overall, the objective of minimizing the negative runtime performance impact of exception handling at high optimization levels is achieved. This constrasts with PA-RISC, where penalties as high as 15% are observed (and in practice 10% is not uncommon). It remains to be seen if this conclusion remains valid as more aggressive optimizations are added to the IA-64 compiler.

The size penalty on IA-64 tends to be higher, at least at optimization levels used during application development. This is due largely to cleanup code, which takes more space than the same information stored in PA-RISC action tables. As usual, there was a space versus time trade off, and this technology definitely favored speed.

The added exception handling code is normally infrequently executed. Modern operating systems do not load code into memory until it is about to be executed. So most of the time, the additional code just consumes disk space, without necessarily increasing the memory footprint of the application. Disk space gets cheaper all the time, so the trade off was a reasonable one.

This code size penalty may be reduced somewhat in the production compilers by a change in the C++ Application Binary Interface (ABI) [6] that is not implemented in the tested compiler. This change reduces the size of a minimal landing pad from 32 bytes down to 16.

The reason for the size reduction at maximum optimization has not been investigated yet. It may indicate a problem with the compiler, such as an optimization being accidentally turned off when exceptions are disabled. Exception handling may also prevent some code-expanding transformations that look less profitable, such as inlining and loop unrolling. It is unclear if these results will persist with a production compiler.

In general, keep in mind that all measurements above were made with a largely prototype compiler, and in a simulator. As our understanding of optimization techniques specifically targetting the IA-64 architecture improves, the results may change significantly.

6. Conclusion

Landing pads offer an interesting alternative to more traditional implementations of C++ Exception Handling. They leave more optimization freedom to the compiler. Many aggressive optimizations can now be performed equally well even in the presence of exception handling code, making applications that require exception handling faster.

One the design objectives of C++ is that you don't pay for features that you don't use. This objective was not met with many exception handling implementations. The IA-64 implementation presented here is one more little step towards that goal.

This design has been shared by Hewlett-Packard with other Unix vendors, and should hopefully become available on a variety of IA-64 platforms as part of the effort towards a common C++ Application Binary Interface for IA-64 on Unix [6].

7. References

[1] *A Portable Implementation of C++ Exception Handling* D. Cameron, P. Faust, D. Lenkov and M. Mehta, Proc. USENIX C++ Conference, August 1992.

[2] *IA-64 Instruction Set Architecture Guide* Revision 1.0, Intel Corporation / Hewlett-Packard Company
http://devresource.hp.com/devresource/Docs/Refs/IA64ISA/Index.html

[3] *IA-64 Register Stack Engine* Chapter 9 of the "*IA-64 Instruction Set Architecture Guide*" [2]
http://devresource.hp.com/devresource/Docs/Refs/IA64ISA/rse.html

[4] *IA-64 Software Conventions and Runtime Architecture* Vers. 1.0, Hewlett-Packard Company
http://devresource.hp.com/devresource/Docs/Refs/IA64CVRuntime.pdf

[5] *Stack Unwinding and Exception Handling* Chapter 11 of "*IA-64 Software Conventions and Runtime Architecture*" [4].

[6] *C++ ABI for IA-64: Exception Handling* Working document, C++ ABI Committee
http://reality.sgi.com/dehnert_engr/cxx/abi-eh.html

[7] *ISO 14882: C++ Programming Language*
http://www.iso.ch

Thanks to M. Brown, D. Cameron, C. Coutant, D. Handly, E. Gornish, D. Gross, R. Ju, and D. Vandevoorde for their contributions to this paper or the techniques it describes.

Application Programming on a Shared Memory Multicomputer

Todd Poynor, Tom Wylegala
HP Laboratories, Palo Alto

The HP Labs MultiComputer Systems (MCS) project is investigating issues involved in writing applications for a shared memory multicomputer, defined as a set of independent computing nodes coupled through access to Global Shared Memory (GSM). MCS is an architecture specification and prototype implementation of a shared memory multicomputer. The prototype platform is based on a commercially available ccNUMA machine comprised of 4 SMP nodes. MCS leverages commodity operating systems with few or no changes; the prototype runs an unmodified Windows NT 4.0 operating system on all nodes, extended through dynamically loaded kernel modules to support multicomputer interfaces.

Shared memory multicomputers hold considerable promise as modular architectures that transcend SMP scaling bottlenecks while preserving SMP-like memory load/store programming models. Many multicomputer platforms today do not fully deliver these benefits because most resources are partitioned and shared memory is limited to inter-node communication, as in a shared-nothing cluster. Those multicomputer platforms that we are aware of that do allow access to global resources have limited support for containing faults from propagating across nodes, leaving multicomputers at a disadvantage when compared to conventional clusters in this regard. A shared-something cluster exposes the risk that faults will propagate over the shared resource to introduce faults in other components. This issue remains a thorny problem in the industry, especially when commodity hardware and operating systems are leveraged.

The MCS project is, in part, an experiment in pursuing fault containment as strong as that of shared-nothing clusters within a shared-almost-everything multicomputer for flexibility of resource allocation and scalability. MCS global applications are a set of processes distributed across the nodes of a multicomputer that cooperate to provide a service. The applications freely employ a variety of global resources on any node. These resources are accessed in a "safe" fashion, detecting and recovering from failures and reconfiguring when the deployment of nodes and applications is changed. This comes at a price of extending commodity operating systems and customizing applications for fault containment, as well as extending commodity hardware platforms to support containment and avoid single points of failure. We examine the challenges of programming in such an environment and investigate support our platform could provide to make the programming task easier.

Consider a thread that reads the identifier of a global mutex from a GSM area, locks the mutex, updates a data structure in the GSM area, and releases the mutex. Possible disruptions in global state include: the mutex identifier may no longer be valid; the GSM hosting the mutex identifier or data structure may fail; the mutex may be found "abandoned" by the previous holder without releasing it; the set of processes and operating systems managing the GSM or mutex may change, requiring a change in global resources; and some other recovery event may be signaled by another thread, requiring the mutex be unlocked before global state may be rebuilt. Part of our task in supporting multicomputer applications is to ease the development and execution of applications in such a dynamic context.

We created a set of C++ classes to facilitate development of global services for a variety of applications and to facilitate multiple global components within one application. Among the services provided are: a framework for initiating, detecting, synchronizing, and handling global system and component failures and recovery events; component membership management, where the set of instances participating in a generation of the global component are agreed upon; and various library functions that automatically perform recovery actions.

To demonstrate our MCS prototype, we modified an existing commercial application to use global resources and to recover from certain software failures. Our primary goal was to demonstrate recovery from an OS crash on one node while shared resources were in use by the crashing node, a relatively high probability failure that exercises both kernel and application layer recovery.

MCS also targets future recovery from a number of hardware failures, including failed memory access resulting from abruptly powered off nodes or malfunctioning memory, some support for which was also prototyped in our application work. Recovery from memory failures on present-day commodity

processors poses some thorny problems in regard to notifying the proper application contexts of failed memory access. On IA-64, for example, these problems include indeterminacy of notification, speculative data prefetches, and advanced instruction retirement prior to completion of memory operations. We propose a model that avoids many of these problems in that the system is not required to match a failed access with the context that issued the request. This places a greater burden upon applications to detect failures and increases the chances that an application will perform unnecessary repair work, but failures should be rare.

Our demonstration application is a Web server file cache that reduces disk I/O by caching local static files in memory (also known as a "reverse proxy"). Both the Apache and Microsoft IIS Web servers are adapted to a common MCS global Web caching component. A mixture of the two Web servers may run in the same multicomputer complex; all will share the same cache. This is not an ideal application for demonstration of multicomputer platforms, as the primary data being shared is read-only and easily replicated and partitioned across shared-nothing servers with little or no inter-node communication. The Web cache application does, nonetheless, allow us to investigate various aspects of the recovery model and performance scalability, using an easily modified technology. Failure and reconfiguration recovery scenarios include:

- A process fails while updating data structures such as hash chain pointers and cached file data reference counts. The hash chain must be rebuilt and references from failed processes cleared.
- Processes enter and leave the global application concurrent with ongoing cache access or recovery, requiring coordinated changes in the global memory areas and perhaps mutexes in use. If processes have left the global application then the associated cached files are removed from the hash table.
- GSM access may fail while building shared state, while traversing hash chains and updating cached file data reference counts and access times, and while sending cached data back to the HTTP client.

We have demonstrated that the application recovers from multiple process failures at a time in each situation where the failure could affect other processes. Application-level recovery from failure occurs within 2 seconds, which suggests competitive performance with application recovery times for a number of the leading cluster failover solutions available today. The MCS project did not reach the point where the complete system recovers from an OS crash, but several individual software components of the system have demonstrated this under prototype conditions. The prototype platform is not suitable for demonstrating recovery from hardware failures, as the interconnect cannot recover from unresponsive nodes.

Although we did not perform extensive performance tuning of the Web cache, we did characterize performance using a workload based on a popular Web server benchmark that serves static content. We consistently obtained 2X performance when resources were doubled by moving from one to two nodes with double the amount of aggregate memory for the cache. We can expect to obtain better than 2X scaling when memory is doubled on a suitable platform because the effective cache size of a shared nothing configuration is not doubled due to duplication of frequently accessed content in both caches. In our trials we saw only an insignificant increase beyond 2X scaling, caused by an excessive GSM access penalty of 12X in the prototype hardware. Experiments substituting local memory for the cache show a 40% performance improvement and linear memory scaling, suggesting that much better performance would be obtained on a more suitable platform.

In addition to researching technology issues related to platform software and hardware, the MCS program examined business issues related to the acceptance of such a platform in the commercial application marketplace, consulting with researchers and developers at a variety of potential business partners. The first major concern voiced by the ISVs was the standardization of the global APIs: (1) that the vendor of the host operating system certify the global APIs; (2) that all shared memory multicomputers based on Windows NT share the same global APIs; (3) that all shared memory multicomputers, even those based on other operating systems, share some similarity in API structure. The second major concern was the approach taken to achieve scalability across multiple nodes, which may be at odds with their existing strategy. The last major concern relates to the added difficulty of achieving high availability under the MCS failure model, a discipline not required on SMP platforms.

To help address the difficulties of attracting ISVs to multicomputer platforms, we began work on the recoverable component framework. We also planned for a developer kit, to include various global status display and modification tools, as well as application and kernel driver debugging tools adapted for concurrent, multinode debugging.

More information on our work may be found in an HP Labs Technical Report titled "Application Programming on a Shared Memory Multicomputer", to appear at http://www.hpl.hp.com/techreports/ .

Meeting Performance Goals with the HP-UX Workload Manager

(an extended abstract)

Indira Subramanian
Hewlett Packard Co.
Cupertino, CA
indira@cup.hp.com

Cliff McCarthy
Hewlett Packard Co.
Richardson, TX
mccarthy@rsn.hp.com

Michael Murphy
Hewlett Packard Co.
Richardson, TX
mmurphy@rsn.hp.com

The HP-UX Workload Manager helps workloads meet user-specified performance goals by dynamically adjusting their access to resources such as CPU. We implemented this workload manager as a part of a *feedback control system*, using existing resource control and performance instrumentation infrastructure.

1. Introduction

Successful consolidation of multiple workloads on to a single server demands that users be guaranteed consistent levels of workload performance. Users should be able to define Service Level Objectives (SLO), specifying the performance goals they seek and their relative importance. To achieve target performance consistently, applications' access to resources such as CPU and memory must be adjusted automatically.

Workload managers can be classified in to two categories. An *entitlement-based* manager allocates resources based on a specification of resource entitlements. *Goal-based* workload managers adjust the resources allocated to a workload, based on a specification of performance goals.

Entitlement-based and goal-based workload managers have been supported in some commercial and experimental systems. Several UNIX OS vendors implement entitlement-based resource managers[9, 5, 4], which do not use any feedback mechanism to meet performance goals. IBM's OS/390 goal-based workload manager (WLM) employs extensive instrumentation to gather detailed information about an application's resource needs, and adjusts resource allocations[1]. Adjusting the use of system resources to meet response time goals has been in wide use in Transaction Processing (TP)[2]. Several experimental transaction processing systems have exploited feedback mechanisms to meet response time goals[7, 8].

The HP-UX Workload Manager is distinct from the systems discussed above that also use feedback control to adjust resources. First, the workload manager does not monitor workload performance directly. Instead, it receives a workload's performance data through an API from a performance monitor created by the application provider (or system administrator). Second, the workload manager uses a simple proportional controller to determine the resources that must be allocated to a workload. Third, the workload manager has been designed to take advantage of the existing infrastructure of tools that includes Process Resource Manager (PRM), and the Event Monitoring Service (EMS, which raises an alarm when a performance goal is not being met). Fourth, unlike TP monitors, which handle workload management exclusively for transaction processing systems, the HP-UX Workload Manager can handle a wide variety of application workloads.

2. Background

The HP PRM (Process Resource Manager), enables a system administrator or the workload manager to control the resources allocated to a workload[3, 4]. PRM is tightly integrated with the HP-UX kernel, and is supported through enhancements to certain HP-UX system calls and commands. PRM monitors resource usage, and it can guarantee a minimum entitlement of CPU, memory, and disk bandwidth available to a group of processes (a PRM group, or a resource group). PRM can also enforce capping (upper bound) of CPU and memory allocated to a resource group. The PRM scheduler selects (to run) a group with larger CPU entitlement, more frequently than other groups. The PRM interface allows an administrator to specify the group to which a user belongs. All processes in a group share the resources assigned to that group. Within a group, standard HP-UX resource management policies are applied.

The HP-UX Workload Manager enhances PRM in two ways. First, by giving each workload only the resources that are needed to meet its goal, excess capacity is shared efficiently across workloads. Second, a higher priority workload gets the resources it needs, before the needs for lower priority workloads are met. This priority is distinct from the UNIX process priority. WLM uses this priority to resolve resource entitlement conflicts, when the resource needs of all the workloads cannot be met. A PRM group with higher CPU entitlement is selected to run more frequently than other groups. The standard HP-UX scheduler employs the UNIX process priority to schedule processes from the selected PRM group.

3. System Overview

The key to developing a successful workload manager is recognizing the fact that it is part of a closed feedback loop control system [6]. The output of the control system is the new set of resources (*entitlement value*) that must be allocated for the workload. To set the new entitlement value, the workload manager enlists PRM. Inputs to the WLM controller are the SLO, priority, and the measured performance. The system administrator specifies the goals and priorities for the workloads on the system, in a simple text file (the WLM configuration file). To obtain performance measurements, the system administrator specifies a performance monitor program. Each workload with a SLO goal, must have a performance monitor associated with it. The WLM daemon starts up this performance monitor process, which communicates subsequently with WLM through a simple API. The WLM controller wakes up periodically, subtracts the goal value from the most recent performance number, multiplies this result by a proportionality constant (`cntl_kp`), and adjusts the entitlement by this amount.

For the workload's performance to converge towards the goal, we require that the relationship between entitlement and performance be monotonic. We choose to ignore a variety of other factors that might influence performance. However, this oversimplification allows us to analyze the characteristics of the workload in useful ways by focusing on the average behavior.

4. Tuning the System

To achieve effective automatic control of workload performance with WLM, three tunable parameters are supported: `wlm_interval`, `cntl_kp`, and `cntl_margin`. `Wlm_interval` controls how often WLM wakes up, checks performance data, and makes entitlement adjustments. WLM makes the change request based only on the latest value reported by the performance monitor. `Cntl_kp` controls how big an adjustment is made (to the entitlement) in response to a deviation from the service level goal. This tunable is discussed in detail in the next paragraph. `Cntl_margin` is used to specify a safety margin around the service level. A safety margin is used to decide when operators are to be notified.

Eight factors help determine a suitable starting value for `cntl_kp`. These factors and their values represent the workload characteristics and the desirable rate of convergence to the workload's performance goal. For this discussion, we consider a transaction processing workload, where the performance goal is that the average completion time for a transaction be under `t` seconds. Four of the factors that determine `cntl_kp` are upper bound (U seconds) and lower bound (L seconds) on service levels, and the corresponding resource entitlements Eu (% CPU) and El (% CPU) respectively. The fifth factor is G (seconds) , the goal value for SLO. We also need to consider the rate of convergence, that is, how long we are willing to wait to get within a certain range of the goal. The range A is expressed as a fraction of G, and T (seconds) specifies the time to get within this range. The eighth factor is J – the average number of seconds between entitlement changes for the workload. Given these parameters, a reasonable initial estimate for `cntl_kp` can be obtained. The expression for calculating `cntl_kp` written using the natural logarithm (base `e`):

$$\texttt{cntl\_kp} = -ln(\frac{AG}{\|U-L\|})\|\frac{J(Eu-El)}{T(U-L)}\|$$

Because of the assumptions made in the calculations that led to this formula, this initial value of `cntl_kp` may need fine-tuning to produce the desired behavior. However, it serves as a reasonable starting point. This initial value must be adjusted based on actual measurements of the time it takes to converge to the response time goal.

5. Summary

The HP-UX Workload Manager employs a feedback control system to dynamically adjust CPU resources allocated to a workload. A future version will handle other resources including memory and I/O. The workload manager uses the existing infrastructure of resource control, performance instrumentation, and event monitoring tools.

References

[1] J. Aman, C. Eilbert, D. Emmes, B. Yocom, and D. Dillenberger. Adaptive algorithms for managing distributed data processing workload. *IBM Systems Journal*, 36(2), 1997.

[2] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Number ISBN 1-55860-190-2. Morgan Kaufmann Publishers, San Francisco, CA, 1993.

[3] Hewlett-Packard Company. *HP-UX Process Resource Manager*, May 1999. Technical White Paper.

[4] Hewlett-Packard Company. *HP-UX Process Resource Manager User's Guide*, December 1999.

[5] International Business Machines Corporation. *AIX 4.3.3 Workload Manager*, February 2000. Technical White Paper.

[6] B. C. Kuo. *Automatic Control Systems*. Number ISBN 0-13-304759-8. Prentice Hall, Upper Saddle River, NJ, 1995.

[7] Markus-Sinnwell and A. C. Konig. Managing distributed memory to meet multiclass workload response time goals. In *15th. International Conference on Data Engineering*, March 1999.

[8] E. Rahm. Goal-oriented performance control for transaction processing. In *9th. ITG/GI MMB97 Conference*. VDE-Verlag, 1997.

[9] Sun Microsystems. *Solaris Resource Manager [tm] 1.0: Controlling System Resources Effectively*, 2000. http://www.sun.com/software/white-papers/wp-srm.

Dynamic Memory Management with Garbage Collection for Embedded Applications

Roberto Brega and Gabrio Rivera
brega@ifr.mavt.ethz.ch and rivera@inf.ethz.ch

Swiss Federal Institute of Technology Zurich (ETHZ)
CH–8092 Zurich

Abstract

A software system can be called a *safe-system* with respect to memory, when it supports only strong-typing and it does not allow for the manual disposal of dynamic memory [2]. The first aspect guarantees that untyped, potentially dangerous operations are caught by the compiler or by run-time checks. The second issue is solved by the utilisation of an automatic memory reclamation scheme, i.e. a garbage collector.

In this paper we argue that the careful choice of the programming language, along with an automatic memory reclamation scheme can optimise memory usage, while ensuring that many of the logical errors related to memory can be avoided.

1 Introduction

Implementors of modern embedded machines must face two opposing tendencies. On the one hand, the system should behave like a modern desktop system, with its latest standards, the support for inter-networking, and ease-of-use. On the other hand, unlike a modern desktop system, it is not allowed to fail and must be able to run on tight memory conditions. Therefore implementors will often need to trade-off features for reliability.

This same issue was faced by the Institute of Robotics of the ETH Zurich, Switzerland and by Mechatronic Research Systems, an ETH spin-off company specialised in robotic control systems, as they were asked to realise state-of-the art control solutions for high-end mechatronic products by two Swiss companies.

Meyco Equipment AG needed a robot controller for a redundant manipulator spraying liquid concrete for tunnelling work. Besides the inherent difficulty of controlling the redundant, hydraulically actuated manipulator, the system needed to provide remote diagnostics through standard web services and remote controlling from an embedded console through TCP connections. Similar requirements were to be addressed for a fully autonomous fork-lift truck used in a warehouse of a Swiss manufacturing plant. The two machines needed to have months of up-time, while providing complex software services that could lead to small but inevitable memory leaks, thus having their long-term reliability undermined.

Unwilling to compromise on run-time safety, the Institute of Robotics chose to attack these problems at their roots, by deploying its in-house developed real-time operating system XO/2 [1].

2 The Role of Strong-Typed Programming Languages in Safe Systems

System components are the tools of a software engineer. The safer the tools, the more reliable is the system. It is well-known that languages, or more precisely proper language paradigms and type systems, can do a lot to help programmers. Strangely enough, despite the existence of better alternatives, a lot of safety-critical software gets implemented by means of programming languages that do a poor job of ensuring static or dynamic safety. In those cases, tools are used for uncovering errors that should not have been made possible in the first place, such as array indexes out of range, dangling pointers, casting errors, and memory leaks.

A type-system, as can be found in strong-typed languages, can be seen as the primary method for annotating all entities of a given program. The programmer can no longer tinker with memory addresses and he is forced to play by the rules of the language, in order that his program correctly compiles and is not trapped in a run-time assertion. The assertions bound to program entities enforce a more well-thought structural design, thus they can help in diminishing the chances of deep conceptual errors in the application as well.

3 Automatic Disposal of Dynamic Memory

The central knowledge of all references that exist for a particular object becomes hard to maintain as the dynamic loading of extensions increases. Even worse, it becomes impossible for a programmer to keep track of references in a safe way when the programming language does not impose restrictions on the passing and copying of references. This brings us to the conclusion that in a dynamically extensible system, explicit deallocation of objects is not feasible. Failing to realise this

introduces a new class of run-time problems, such as dangling references and memory leaks.

The only safe possibility for object disposal is by means of a system-wide mechanism performing automatic storage reclamation: a *garbage collector*. A garbage collector defines the liveness of heap objects by their reachability, starting from a working set of global and local references: an object is live only when there is some path that reaches it; on the other hand, an object is not live when no path can be found that reaches the object. When an object is not reachable, it can be disposed of safely.

4 The XO/2 Heap-Manager

XO/2 follows the guidelines presented in sections 2 and 3 in its dynamic memory manager: Each allocated object is bound to a type, and the responsibility of its reclamation is left to the system-wide garbage collector. The heap manager in XO/2 implements a *mark-and-sweep* garbage collector. This scheme has been chosen because of the possibility of an incremental implementation, and its non-destructive behavior during conservative-marking of the procedure's activation frames [3].

The algorithm consists of two distinct phases. The first one, the *marking phase*, traverses the objects graph, by starting from a well-defined *root-set*. The root-set is made up of the global pointers declared in a module variables-block, and all of the local pointers present in the processes' stacks and registers. Every traversed object is marked and their descendants traversed until every reachable object in the heap space has been marked. During the second phase, the *sweep-phase*, the collector disposes of the objects in the heap that have not been traversed.

The pre-emptive, real-time nature of the XO/2 task scheduling explicitly requires the collector to be incremental, interruptible and with a low-overhead with respect to the concurrent running programs, without blocking or delaying accesses to heap-objects.

The traversal phase (marking) of the collector is very sensitive to outside changes of the objects graph brought by pointer operations. Pointer assignment operations such as $p:=q$ (where p and q are pointers of the same type or where q is a subtype of p) would invalidate the marking of the graph performed by the collector during the heap traversal.

In order to handle these operations, the compiler generates code for notifying the collector that a change in the graph has occurred. The newly injected code produces only a 1% overhead.

5 Space and Time Considerations

It is of outmost importance that the collector can be started when a low-memory condition occurs. Although memory-efficient traversal algorithms exist—such as the pointer-reversal descent—they are not suitable for pre-empted operation. Another possibility is the classic stack-based traversal. Unfortunately the stack-space needed for marking the heap is proportional to the depth of the graph: When the heap grows too much, the amount of memory available to the traversal decreases accordingly, thus potentially inhibiting the completion of the marking phase.

We devised a scheme that allows the collector to completely mark the heap without allocating additional memory. Consequently, the collector can always complete marking and sweeping regardless of the amount of free memory.

6 Real-World Experiences

The autonomous fork-lift truck and the RoboJet manipulator have been benchmarked against memory usage under different memory load conditions.

We inspected two different heap configurations: the first one with a 512 KB heap-size (basic product functionality), the second one with a 4 MB heap-size (continuous networked monitoring enabled), both of them with an average objects' size of 128 bytes. For completing one mark-and-sweep pass, while the full tasks constellation was running, the average collection time for the 512 KB heap-size amounts to 8 ms, and that for the 4 MB heap-size to 24ms. The tests show that the time needed for a complete collection pass remains low for heap-sizes that are common for our mechatronic applications.

7 Conclusion

As the list of requirements for embedded machines grows, application programmers can find themselves overwhelmed. A careful choice of the programming language, along with a system-wide mechanism for automatic storage reclamation provide invaluable help in keeping the system complexity low. We believe an incremental garbage collection mechanism, supporting transparent execution with respect to real-time tasks should find its way into today's mechatronic products.

References

[1] R. Brega. A real-time operating system designed for predictability and run-time safety. In *Proceedings of The Fourth International Conference on Motion and Vibration Control (MOVIC)*, pages 379–384, Zurich, August 1998.

[2] C. Szyperski and J. Gough. The role of programming languages in the life-cycle of safe systems. *Second International Conference on Safety Through Quality (STQ'95)*, Kennedy Space Center, Cape Canaveral, Florida, USA, October 1995.

[3] Paul R. Wilson, Uniprocessor Garbage Collection Techniques, *Submitted to ACM Computing Surveys*.

A fast implementation of DES and Triple-DES on PA-RISC 2.0

Francisco Corella
Hewlett Packard Co.

1. Introduction

Encryption is an essential tool for protecting the confidentiality of data. Network security protocols such as SSL or IPSec use encryption to protect Internet traffic from eavesdropping. Encryption is also used to protect sensitive data before it is stored on non-secure disks or tapes.

Encryption, however, is computationally expensive. A computer server that must encrypt data for thousands of clients before sending it over the network can easily become crypto-bound. The capacity of the server is then determined by the speed at which it can perform encryption. This is especially the case when slow encryption protocols such as the Digital Encryption Standard (DES) or Triple-DES are employed. Since DES and Triple-DES are very widely used, it is important to optimize the performance of these algorithms.

We describe an implementation of DES and Triple-DES in PA-RISC 2.0 assembly language that outperforms other practical (non bit-sliced) implementations by large margins. It is based on a technique due to Eli Biham of the Technion that takes advantage of 64-bit registers, with substantial improvements developed at Hewlett Packard.

We assume that the reader is familiar with the details of DES and Triple-DES, which are described in [1].

2. Earlier software implementations

Most software implementations of DES and Triple-DES are written for machines having 32-bit registers. In 1997, Eli Biham published an implementation, written in C, specifically targeted for the 64-bit Alpha architecture [2].

Biham took advantage of the 64-bit register width as follows. He stored the two block halves that each round of DES operates on in two separate 64-bit registers. However, instead of storing them in their standard 32-bit format, he stored them in the 48-bit format that results from applying the expansion permutation to a 32-bit array, with zeros in the twelve remaining bit positions. Each round then proceeds as follows. The right half, which is already in expanded form in a 64-bit register, is xored with the subkey, which is also contained in a 64-bit register. Then the resulting 48-bit array is divided into eight groups of six bits, each of which is used as the index into an S-box.

Biham also changed the format of the S-boxes. He turned each S-box entry into a 64-bit array, derived from the 4-bit array of the original S-box by the following procedure. First the 4-bit array is placed in its proper position within a 32-bit (unexpanded) block half, the other bit positions in the array being filled with zeros. Then the 32-bit permutation is applied to the 32-bit array. Finally, the expansion permutation is applied to this 32-bit array, producing a 48-bit array, which is completed with zeros in the twelve remaining bit positions. Biham's algorithm does an S-box look up as a straightforward DES implementation would, but the look up produces a 64-bit array, mostly filled with zeros, rather than a 4-bit array. After the eight S-boxes have been looked up, the eight resulting 64-bit results are ored together and the result is then xored with the left half, which is also stored in a 64-bit register in the expanded 48-bit format.

With this implementation, Biham achieved a throughput of 46 Mb/s for DES and 22 Mb/s for Triple-DES on a 300 MHz Alpha 8400 processor. He compared this performance to that of Eric Young's libdes library on the same machine, which presumably does not take advantage of the 64-bit register width. The performance of libdes was quoted by Biham as 28 Mb/s for DES.

Recently, however, other 32-bit implementations have achieved better performance. The popular BSAFE cryptographic toolkit of RSASecurity, can serve as a good benchmark for comparison purposes. At the 1999 RSA conference, substantial performance improvements for several BSAFE algorithms were announced. The new performance figures quoted for DES and Triple-DES were 39.2 Mb/s and 15.2 Mb/s respectively, on a 233 MHz Pentium II.

In the same paper [2], Biham also describes a bit-sliced implementation, which has higer throughput. However, a bit-sliced implementation is not practical because application software would have to be restructured to take advantage of the 64-way parallelism that yields the increased throughput.

	Young's 32-bit implementation	Biham's 64-bit implementation	BSAFE 32-bit implementation	Our 64-bit implementation
Throughput	28 Mb/s	46 Mb/s	39.2 Mb/s	183 Mb/s
CPU frequency	300 MHz	300 MHz	233 MHz	550 MHz
Relative speed	686 clocks/block	417 clocks/block	380 clocks/block	192 clocks/block

Table 1. Comparison of four DES implementations (CBC mode)

3. Our implementation

Biham provides an instruction count for his standard DES implementation. Processing one DES block takes 634 instructions. A majority of these instructions (61%) are concerned with S-box look-up. Looking up an S-box takes three instructions, and this is done 8 times per each of the 16 rounds, for a total of 16x8x3 = 384 instructions.

We have made two key improvements to Biham's implementation. First, we have grouped the eight S-boxes into four pairs of boxes, and merged the two 6-input boxes in each pair into a single 12-input box.[1] The resulting four double S-boxes occupy 1 MB of memory.

Then, writing the code in PA-RISC 2.0 assembly language [3], we have reduced the number of instructions needed to look up an S-box from 3 to 2. An S-box look-up is performed by the following two-instruction code fragment:

```
EXTRD,U     gr1,pos,12,gr2
LDD,S       gr2(gr3),gr4
```

The first instruction extracts an unsigned 12-bit value from bit position pos of general register gr1, and places it in general register gr2. In the implementation, the general register gr1 contains the result of xoring the expanded right half with the subkey for the round. The immediate value pos selects one of the four groups of 12 bits that are used to look up the four 12-input S-boxes.

The second instruction shifts left by three positions (i.e. multiplies by 8) the value contained in gr2, adds the result (a displacement) to the contents of general register gr3 (a memory address), and loads the 64-bit word stored at the resulting memory address into general register gr4. In the implementation, gr3 contains the base address of the 12-input S-box, gr2 contains the 12-bit index into the S-box, and gr4 receives the 64-bit entry read from the referenced S-box entry.

The cumulative result of these two improvements is that S-box look up only requires a total of 16x4x2 = 128 instructions, thus saving 256 instructions. A variety of other improvements saved another 108 instructions, resulting in a count of only 270 instructions for a DES computation. Table 1 shows the performance achieved by our DES implementation, comparing it to that of the other implementations mentioned above. Our Triple-DES implementation takes only 471clocks/block (74.8 Mb/s on a 550 MHz CPU), while Biham's and BSAFE take 873 and 981 clocks/block respectively.

Our implementations have been incorporated in the IPSec/9000 product available with HP-UX, and are currently being incorporated in a cryptographic toolkit.

[1] This idea is due to Peter Markstein.

Acknowledgements

Peter Markstein contributed the idea of merging two 6-input S-boxes into a single 12-input box. Ruby Lee provided motivation for the project and drew attention to Eli Biham's paper [2].

References

[1] FIPS 46-3, Data Encryption Standard (DES), National Institute of Standards and Technology (NIST), http://csrc.nist.gov/cryptval/des.htm.

[2] Eli Biham, *A Fast New DES Implementation in Software*, Fast Software Encryption 4, Haifa, Israel, January 1997. Available from Eli Biham's home page, http://www.cs.technion.ac.il/~biham/.

[3] G. Kane, *PA-RISC 2.0 Architecture*, Prentice Hall, 1996.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

Supporting Members of the USENIX Association:

Addison-Wesley
Earthlink Network
Edgix
Interhack Corporation
Interliant
JSB Software Technologies
Lucent Technologies
Macmillan Computer Publishing, USA
Microsoft Research
Motorola Australia Software Centre
Nimrod AS
O'Reilly & Associates Inc.
Performance Computing
Sendmail, Inc.
Smart Storage, Inc.
Sun Microsystems, Inc.
Sybase, Inc.
Syntax, Inc.
UUNET Technologies, Inc.
Web Publishing, Inc.

Supporting Members of SAGE:

Collective Technologies
Deer Run Associates
Electric Lightwave, Inc.
ESM Services, Inc.
GNAC, Inc.
Macmillan Computer Publishing, USA
Mentor Graphics Corp.
Microsoft Research
Motorola Australia Software Centre
New Riders Press
O'Reilly & Associates Inc.
Remedy Corporation
RIPE NCC
SysAdmin Magazine
Taos: The Sys Admin Company
Unix Guru Universe

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA.
Phone: 510-528-8649. Fax: 510-548-5738. Email: *office@usenix.org*.

ISBN 1-880446-15-4